# Runtime Scheduling, Allocation, and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 905057, 32 pages
http://dx.doi.org/10.1155/2013/905057

*Research Article*

# Runtime Scheduling, Allocation, and Execution of Real-Time Hardware Tasks onto Xilinx FPGAs Subject to Fault Occurrence

**Xabier Iturbe,[1,2] Khaled Benkrid,[2] Chuan Hong,[2] Ali Ebrahim,[2] Tughrul Arslan,[2] and Imanol Martinez[1]**

[1] *Embedded System-on-Chip Group, IKERLAN-IK4 Research Alliance, 20500 Mondragón, Spain*
[2] *System Level Integration Group, The University of Edinburgh, Edinburgh EH9 3JL, UK*

Correspondence should be addressed to Xabier Iturbe; xiturbe@ikerlan.es

Received 3 May 2012; Revised 2 October 2012; Accepted 3 October 2012

Academic Editor: René Cumplido



This paper describes a novel way to exploit the computation capabilities delivered by modern Field-Programmable Gate Arrays (FPGAs), not only towards a higher performance, but also towards an improved reliability. Computation-specific pieces of circuitry are dynamically scheduled and allocated to different resources on the chip based on a set of novel algorithms which are described in detail in this article. These algorithms consider most of the technological constraints existing in modern partially reconfigurable FPGAs as well as spontaneously occurring faults and emerging permanent damage in the silicon substrate of the chip. In addition, the algorithms target other important aspects such as communications and synchronization among the different computations that are carried out, either concurrently or at different times. The effectiveness of the proposed algorithms is tested by means of a wide range of synthetic simulations, and, notably, a proof-of-concept implementation of them using real FPGA hardware is outlined.

## 1. Introduction

Dynamic Partial Reconfiguration (DPR) permits to adjust some logic resources on FPGA chips at runtime, whilst the rest are still performing active computations. During the last few years, DPR has become a hot research topic with the objective of building more reliable, efficient, and powerful electronic systems. Indeed, DPR is the enabling technology for a new computing paradigm which combines computation in time and space [1–3]. In Reconfigurable Computing (RC), a battery of computation-specific circuits ("hardware tasks") is swapped in and out of the FPGA on demand to hold a continuous stream of input operands, computation, and output results. Multitasking, adaptation, and specialization are key properties in RC, as multiple swappable tasks can run concurrently at different positions on chip, each with custom data paths for efficient execution of specific computations.

Besides task switching, DPR makes it possible to deal with the increasing fault rate currently observed in advanced electronic devices [4] in a more reliable way than ASICs do. While the latter, traditionally, simply tolerates the fault effect by using redundancy, FPGAs permit to route around damaged resources on-the-fly, keeping the system fault-free at all times [5].

However, DPR penetration in the commercial market is still testimonial, mainly due to the lack of suitable high-level design tools to exploit this technology. Indeed, currently, special skills are required to successfully develop a dynamically reconfigurable application.

In light of the above, we aim at bridging the gap between high-level application and low level DPR technology by developing Operating System-(OS-) like support for high-level software-centric application developers. Our solution is named as R3TOS, which stands for Reliable Reconfigurable Real-Time Operating System. R3TOS defines a flexible infrastructure for coordinately and reliably executing RC applications under real-time constraints using partially reconfigurable Xilinx FPGAs [6].

In this article we first describe a novel way to harness the internal reconfiguration mechanism of modern FPGAs to perform intertask communications and synchronization

regardless of the physical location of tasks on chip. Specifically for this scenario, we propose a novel EDF-based scheduling algorithm, two novel task allocation heuristics (EAC and EVC), and a novel task allocation strategy (called *Snake*). A preliminary description of EAC and EVC allocation heuristics has been presented in [7], while the *Snake* task allocation strategy has been firstly proposed in [8]. This article provides an extended explanation of these approaches as well as new experimental results in order to gain a better understanding of them. Furthermore, the article briefly describes how our algorithms have been implemented in a real Xilinx Virtex-4 FPGA.

The remainder of this article is organized as follows. After stating the reconfigurable scenario defined by DPR-enabled Xilinx FPGAs in Section 2, Section 3 explains the related state-of-the-art. Then, an overview on the R3TOS approach is provided in Section 4, followed by a description of the used scheduling and allocation heuristics and algorithms in Sections 5 and 6, respectively. In Section 7, the obtained simulation results are outlined, while Section 8 outlines a real implementation of these algorithms using real FPGA hardware. Finally, conclusion remarks are summarized in Section 9.

## 2. Xilinx FPGAs-Defining Reconfigurable Scenario

An FPGA can be modelled as an architecture with two layers: the *functional layer*, which contains the physical resources used to perform computation, and the *configuration layer*, which controls the configuration of the functional layer.

Although the work described in this paper is focussed on Xilinx Virtex-4FPGAs, we plan to target modern Virtex-6 and Virtex-7 FPGAs in the future, whose architecture is not fundamentally different from Virtex-4, for the purpose of our study.

The functional layer of Virtex-4 devices includes a regular array of Configurable Logic Blocks (CLBs), Input-Output Blocks (IOBs), clock resources, and some special resources such as BRAM memories and DSP48s. CLBs, organized in a regular array, are the most abundant resources and provide logic, arithmetic, data storage, and data shifting functions. On the other hand, IOBs, DSP48s, and BRAMs can be seen as the heterogeneous resources embedded in the middle of the homogeneous CLB array, being organized in columns which span the whole height of the device. Newer devices accentuate the trend for including more columns of heterogeneous resources. All FPGA resources are one-to-one connected by means of a vast amount of programmable wires, allowing for an incredible capability of data movement among registers and memory elements.

The configuration layer consists in a memory, which stores the bitstream defining the functionality implemented by the physical resources in the functional layer as well as their interconnections. The configuration memory is organized in configuration frames of 1312 bits each, and the frame addressing scheme includes information related to the position of the physical resources they configure. Specifically, each frame configures a resource column spanning the whole height of a fabric clock region within the chip. For Virtex-4 devices, each clock region is 16 CLBs, 4 BRAMs, 8 DSP48s, or 32 IOBs high. The CLB, DSP48, and IOB frames include configuration information related to both the resources themselves and the associated routing wires to these resources. BRAMs however include separate configuration frames for the memory content data and the associated wires; that is, while 64 frames are necessary for configuring the 72 Kb memory content data of 4 BRAMs, only 20 frames are used for configuring their associated routing resources. Each BRAM content frame stores 256 bits of information and 32 bits of parity of each of the 4 memories mapped to that frame.

The use of SRAM technology in the configuration memory of Xilinx FPGAs permits to download new configuration data at runtime, which is the key for DPR. Modern Xilinx FPGAs ease the access to this memory by including an Internal Configuration Access Port (ICAP), which makes up the interface between both layers of the FPGA. However, the existence of a single ICAP makes the reconfiguration process sequential, although the circuitry previously configured on the FPGA works simultaneously. Virtex-4 ICAP is 32-bit width and is able to operate up to 100 MHz, allowing a theoretical bandwidth of 400 MB/s.

In summary, an FPGA can be seen as a two-layered device, with the capability of performing on-demand computation, true multitasking, and huge internal bandwidth in its functional layer, yet sequential and limited communication bandwidth with its underlying configuration layer. Therefore, the efficacy of RC depends on the capability for masking sequential transfers to the FPGA-based "computation cache" with the advanced parallel computation of its functional layer.

## 3. Related Work

In [9], the authors propose two preemptive scheduling algorithms for periodic tasks: EDF Next Fit (EDF-NF) and Merge-Server Distributed Load (MSDL). In EDF-NF, the task with the closest deadline that fits in the FPGA is scheduled first, while, in MSDL, tasks are successively merged into servers, which are then scheduled sequentially using EDF.

In [10], the authors propose two nonpreemptive scheduling algorithms for sporadic hardware tasks: *horizon* and *stuffing*. These algorithms keep track of future releases of area when the executing tasks finish, and, upon a new task arrival, they simulate the future state of the FPGA to check whether there will be sufficient adjacent free area to allocate the task before its deadline expires. If not, the task is rejected. In [11], the authors propose a remedy to solve a limitation detected in stuffing; namely, it always allocates tasks on the leftmost edge of the free area, achieving better results. In *classified stuffing*, the tasks with a high ratio between area and execution time are placed starting from the leftmost edge of the free area, while the tasks with a low ratio are allocated on the opposite way. Even better results are reported when applying stuffing over a time window [12]. Based on this, the same authors propose the Compact Reservation (CR) algorithm which is

aimed at reducing the complexity of window-based stuffing [13]. A similar approach is presented in [14], where the authors propose a nonpreemptive algorithm, called *one level look ahead*, which delays the allocation of hardware tasks with the objective of reducing the fragmentation on the device.

All of the algorithms presented above neglect the reconfiguration port exclusiveness and latency of current FPGAs; that is, they assume that reconfiguration does not take time, and, therefore, they are not suitable to be implemented using real hardware. The next generation of scheduling algorithms does consider this restriction.

In [15], the authors propose to schedule access to the reconfiguration port of the FPGA based on the allocation deadlines of the tasks. They port traditional real-time scheduling algorithms for monoprocessors to reconfigurable hardware, that is, preemptive EDF and DM.

In [16], the authors propose to schedule the reconfiguration port access with the objective of optimizing the FPGA utilization, but they do not consider any real-time constraints for the tasks. An interesting idea proposed in this work is the possibility of reusing the already configured tasks on the device, even when they implement only a part of the total functionality required: computation can start in these tasks, while the task that implements the complete functionality is being configured, and once the latter is ready, the partially processed data can be transferred to it. In [17], the same authors propose the so-called 3D Total Contiguous Surface (3DTCS) heuristic that is intended to avoid task placements that will be an obstacle for other incoming tasks in the future. 3DCTS computes the total contiguous surface of the computation volume defined by a given task, that is, the occupied area in time, with the computation volumes of other executing tasks and with the device's boundaries. Therefore, higher 3DTCS value will result in more compaction in space and time.

A third generation of more advanced scheduling algorithms, which are aware of task dependencies and data communications, is currently being developed. A example can be found in [18], where the authors specifically consider data communications between the hardware tasks and external devices. In [19], the scheduling decisions are made using static priorities that are assigned based on the amount of communicating tasks. Finally, in [20], a reconfiguration-aware heuristic scheduler is proposed, which is aimed at exploiting configuration prefetching, task reuse, and anti-fragmentation techniques.

Most of the research efforts carried out up to date in the task allocation field assume a 2-dimensional area model and consider the tasks to be relocatable rectangles that can be placed anywhere on the FPGA device.

The pioneering work described in [21] proposes to Keep track of All the Maximal Empty Rectangles (KAMER) or only of the Non-overlapping Empty Rectangles (KNER) in the device. Note that tasks can be allocated on these empty rectangles without overlapping other tasks already allocated. When allocating a new task in an empty rectangle, some area-fitting heuristics (e.g., Best-Fit, First-Fit) are used to decide whether the rectangle has to be split vertically or horizontally.

In [22], the authors propose to use a hash matrix in which every entry consists of a pointer to a list of the MERs of the corresponding size. Again, area fitting heuristics are used to select in which MER to allocate a task. In light of achieving a better performance, they propose to update the hash matrix while the task is allocated.

In [23], the authors propose to keep track of the occupied area instead of the free area, arguing that the amount of empty rectangles grows much faster than the number of occupied rectangles. Besides, they explain how to simplify the allocation problem by shrinking the area of the chip and simultaneously blowing up the placed tasks by half the width and half the height of the task to be allocated. In this work, the Nearest Possible Position (NPP) algorithm is also presented, which is aimed at minimising the routing cost between the tasks that communicate with each other. The latter routing cost is computed based on the Euclidean distance.

In [24], the authors aimed at constructing staircases with the empty area and, finally, use these structures for finding the MERs. Likewise, in [14], the authors propose the Scan Line Algorithm (SLA) for finding MERs.

In [25] a binary tree is proposed, in which each node represents an occupied location of the device and each leaf represents an MER. This means that, when looking for a suitable location to allocate a task, only leaves have to be explored. Moreover, the authors introduce the Routing Aware Linear Placer (RALP) algorithm, which is aimed at allocating the tasks that communicate together on the empty rectangles with the shortest Manhattan distance to minimize routing costs. In [26], the authors propose to use the boundaries of already placed tasks as routing channels.

A completely different approach is described in [27, 28]. In these works the authors propose to manage the empty area perimeter instead of MERs. In [28], a Vertex List Set (VLS) is used to keep the contour information of each free area fragment in the FPGA. In this context, the authors propose the 2-Dimensional Adjacency (2DA) heuristic, whose objective is to allocate the tasks in the positions with the higher contact perimeter with other running tasks or with the FPGA boundaries. The authors state that these positions are the vertices of already running tasks, which are indeed included in the VLS.

This idea is further developed in [29], where the 3Dimensional Adjacency (3DA) heuristic is proposed. The objective of 3DA is to allocate the tasks in the positions with the highest contact surfaces which are formed when prolonging in time the aforementioned contact perimeter; that is, the execution time of the running tasks is also considered. As a result of including the time domain in the analysis, 3DA outperformed 2DA. In fact, 3DA is currently one of the most effective heuristics for efficiently allocating hardware tasks onto reconfigurable devices, being used or serving as inspiration for other approaches in the field (e.g., 3DCTS [17]).

In [30], the authors propose a heuristic to evaluate the fragmentation on the device based on the amount of existing free area fragments and their shape. This heuristic is used to select the best allocation for incoming tasks in order to minimize the overall fragmentation on the device.

In [31], the Multi-Objective Hardware Placement (MOHP) algorithm is presented, which combines some of the previously proposed ideas. In order to allocate the tasks with close deadlines and no communication requirements the First-Fit heuristic is used; that is, tasks are allocated in the first-found location where they fit. Tasks with slack deadlines and no communication requirements are allocated according to [28, 29], and tasks that need to communicate with other tasks are allocated according to [23].

Finally, a number of authors propose to compact the allocated tasks on the FPGA from time to time in a similar way that the hard drive of a computer is defragmented [32–36]. However, defragmentation techniques incur high reconfiguration overhead provoked by extra task relocations.

We note that most of the approaches described herein use a very abstract device model, which, indeed, can be considered incomplete as it does not account for some physical constraints of FPGAs, for example, granularity of reconfiguration. References [37, 38] are some of the only works which consider these issues. In the former work, the authors propose the Least-Interference Fit (LIF) heuristic with the criteria of interfering running tasks as least as possible when allocating new incoming tasks. However, they exclusively address column-based reconfigurable FPGAs, for example, Virtex-II. In the latter work the authors target modern tile-based reconfigurable devices, for example, Virtex-4.

## 4. The R3TOS Approach

In R3TOS, hardware tasks are scheduled in order to meet their computation deadlines and allocated to nondamaged FPGA resources. In short, R3TOS selects at every kernel-tick $t_{KT}$ the most suitable ready task to be executed according to both time and area criteria. In addition, R3TOS envisages a computing framework whereby both hardware and software tasks coexist in a seamless manner, allowing the user to access the advanced computation capabilities of the modern reconfigurable hardware from a software "look and feel" environment. The software tasks are executed on a main CPU, which can be either built-in on the chip (i.e., hard-core) or implemented using standard FPGA resources (i.e., soft-core).

In R3TOS, the control logic to drive the hardware tasks is attached to their own circuitry, making them self-contained and closed structures which are fully relocatable within the FPGA. This is a completely different approach when compared to related state-of-the-art, where the hardware tasks are executed in predefined reconfigurable slots coupled with fixed control logic and connected to a static communication infrastructure to exchange data among them. Therefore, in R3TOS, the FPGA area is kept free of nonnecessary obstacles at all times, for example, static routes, resulting in higher flexibility to allocate the tasks around the damaged resources (improved fault tolerance) and to increase the computation density by compacting the tasks in the chip (improved efficiency). Furthermore, the complexity of the allocation algorithms is simplified as they do not need to be aware of any underlying implementation-related irregularities in the reconfigurable area. Note that a traditional reconfigurable system must preserve the static routes, resulting in additional difficulties which penalize the performance. In addition, since R3TOS relocates the circuitry along the entire device, the switching activity naturally tends to be distributed among all the resources in the chip; that is, it is not concentrated in some specific regions. This makes the device age uniformly, delaying the occurrence of damage [39]. Note that this does not happen in traditional systems where some resources are prone to fail earlier due to intensive use, for example, the resources used to implement the static communication infrastructure.

The Task Control Logic (TCL) includes an Input Data Buffer (IDB), an Output Data Buffer (ODB), and a Hardware Semaphore (HWS) to enable/disable computation [40]. TCLs provide a means to virtually lock physical data and control inputs/outputs of the hardware tasks to logical positions in the configuration memory of the FPGA. Since the TCLs are accessible through the configuration interface whichever memory positions they are mapped to, the allocatability of the tasks is not constrained by the position of the communication interfaces decided at design time anymore. Furthermore, this scheme improves the multitasking capabilities as the number of tasks that can be concurrently executed on FPGA is not limited by the amount of communication interfaces defined at design time.

In R3TOS, the RC application is modelled as a Directed Acyclic Graph (DAG) where *vertices* represent tasks, and *edges* represent intertask communication channels. Vertices produce and consume data from edges, which in turn buffer the data in a FIFO (first-in, first-out) fashion. Note that buffering capability is mandatory as the tasks can be executed at different slots of time. Besides this, synchronization is mandatory in order to coordinate data producers and consumers when accessing the communication channels.

We distinguish between two types of tasks based on their communication requirements. *High-Bandwidth Communication (HBC) tasks* refer to tasks which process a relatively significant amount of data within a relatively short amount of time. On the other hand, *Low-Bandwidth Communication (LBC) tasks* refer to tasks which process a relatively reduced amount of data within a relatively long amount of time.

The TCLs provide support for communications, synchronization, and data buffering. Namely, the TCL delivers the data to be processed from its internal IDB to the associated task and stores the subsequent results computed by the latter in the ODB. Hence, the data buffers of the TCL are functionally equivalent to the FIFO queues commonly inserted in-between data-processing pipeline stages or to the local caches used in traditional processors. A glue logic adapts data from the way it is stored in the buffers to the needs of the tasks and vice versa (e.g., data rate, word length). It is important to note that each hardware task has dedicated access to its data buffer without interfering with the rest of the tasks running on the device, that is, without constraining multitasking. While the data buffers of HBC tasks are implemented using high-density storage resources (e.g., BRAMs), the buffers of LBC tasks are implemented using low-density storage resources (e.g., distributed memory: LUT-RAMs).

In this context, data is transmitted from a producer task $P$ to a consumer task $C$ by copying the content of $P$'s ODB to $C$'s IDB. If possible, the ODB of the producer task $P$ is configured to be the IDB of the consumer task $C$ so that there is no need to relocate any data; that is, data is in the position the consumer task expects to be. Otherwise, R3TOS harnesses the ICAP to establish on-demand "virtual" channels among the hardware tasks through the configuration layer (see Figure 1).

Additionally, in order to speed up the relocation of high amounts of data in HBC tasks, R3TOS can use physical routes to connect BRAMs when there are no obstacles between them, that is, other tasks. The physical routes and the logic to drive the BRAMs, while data is relocated, are grouped together to form Data Relocation Tasks (DRTs). These are always configured before the computing tasks to which they give support with the objective of parallelizing both the subsequent data relocation, which occurs through the functional layer of the FPGA and the computing task allocation, which occurs through the configuration layer. The physical routes offer higher bandwidth than ICAP-based "virtual" channels, not only because the data is read and written at once, but also because the usable clock frequency can be made higher than that of the ICAP.

Therefore, as shown in Figure 1, in R3TOS the tasks perform computation in the functional layer of the FPGA, and intertask communications are carried out, or at least initiated, through the configuration layer. The synchronization needed to coordinate access to data buffers from both layers is provided by the HWS included in the TCL. The HWS acts as the internal reset signal for the task; that is, the task starts computing only when the HWS is enabled, and once it completes the computation, the task itself disables its HWS. Therefore, the HWS is active only while the task is performing active computation, and, hence, it is also used to implement the exclusive access to FPGA resources.

R3TOS targets an event-triggered data-dependant computation model where data exchanges among tasks are carried out only prior to task execution, with the computation thereafter performed atomically. The tasks are triggered when all their input operands are ready to be processed. This functioning enables *temporal isolation* among hardware tasks execution, avoiding most of communication-related problems in RC, such as deadlocks or race conditions. As a result, the system is predictable enough to approach real-time performance.

In order to achieve higher reliability levels, R3TOS can execute redundant instances of the same (critical) hardware tasks in parallel at distinct positions within the FPGA (i.e., spatial redundancy). The resources assigned to a task instance that has computed an erroneous result are kept in quarantine while an exhaustive diagnostic test is carried out on them. The objective of this test is to detect damaged resources, if any.

*4.1. Real-Time Hardware Task Model in R3TOS.* In the area domain, a task $\theta_i$ is considered to occupy an arbitrarily sized rectangular region on the FPGA, which is defined by its width and height, $h_{x,i}$ and $h_{y,i}$, respectively. The type and amount of resources used by a task depend on the computation it performs; for example, a signal processing task will need to use a large amount of DSP48s. The internal architecture of the task, which depends on the location where it was originally synthesized in the FPGA, is described as the succession of the resources it uses column by column, from the leftmost to the rightmost column.

In the time domain, a task $\theta_i$ requires five different phases to complete a computation (see Figure 2).

(1) During the *set up phase*, the task is configured in the FPGA. Previously existing tasks in overlapping positions are deallocated, if any, and a suitable clock signal is routed to the task. This phase requires $t_{A,i}$ units of time to be completed.

(2) During the *input data delivery phase*, the IDB of the task is filled with actual data to be processed. In a general way, the time required to do so, $t_{D,i}$, is proportional to the amount of data to be loaded. When all input data are copied into the IDB, the HWS of the task is enabled.

(3) During the *execution phase*, the input data is transformed into results by the task's circuitry. The combination of temporally isolated tasks and hardware-based deterministic computation leads to predictable timing behaviour. Indeed, a task uninterruptedly completes its computation $t_{E,i}$ units of time after it started, regardless of system workload. However, $t_{E,i}$ is not always fixed and known, for example, iterative calculations with variable number of iterations. In order to deal with these situations the tasks automatically signal their HWS to acknowledge when the results are ready in their ODB.

(4) The *synchronization phase* refers to the polling on the tasks' HWS to detect a computation completion and spans $t_{S,i}$ units of time. While $t_{S,i}$ is fixed and known for tasks with known $t_{E,i}$, that is, equal to the time needed to access the HWS once, the synchronization overhead for tasks with unknown $t_{E,i}$ varies depending on the amount of HWS accesses carried out until the tasks eventually complete their computations.

(5) During the *output result retrieval phase*, which spans $t_{R,i}$ units of time, the results computed by the tasks are finally read from the ODB. Note that this phase may be delayed until the results are required by another task.

R3TOS keeps track of the state of the tasks at runtime, by grouping them into different task queues: *ready*, *executing* and *allocated*. Note that there is no need for a *Setting-up* queue as only one task can be at this state at a time.

When two hardware tasks communicate each other, data must be read from the producer's ODB prior to being copied to the consumer's IDB. That is, the *output data retrieval phase* of the producer task $\theta_j$ is immediately followed by the *input data delivery phase* of the consumer task $\theta_i$. Furthermore, the latter two phases are to be preceded by the *set up phase* of the data consumer task. When merging these three phases

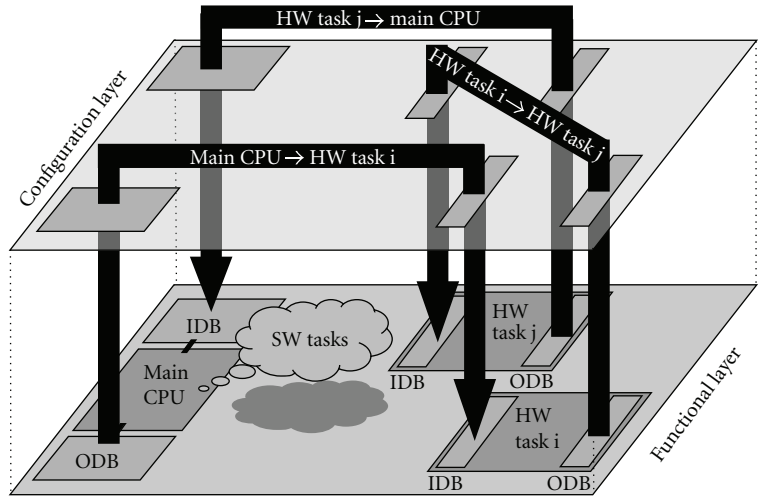



Figure 1: Hardware-software intertask communications using ICAP-based "virtual" channels.

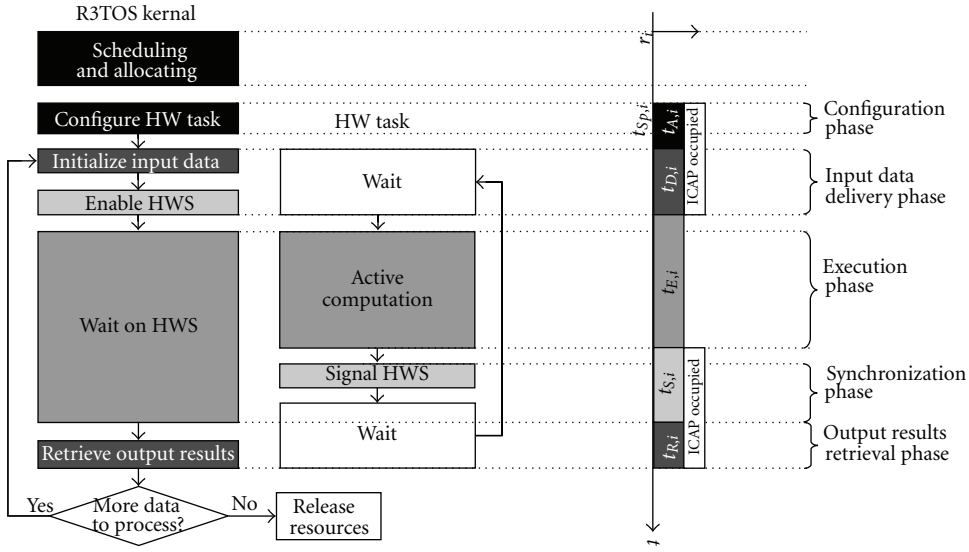



Figure 2: Execution phases of a hardware task in R3TOS.

with the *synchronization phase* of the data producer task $\theta_j$, a single *ICAP access period* is formed for each task $\theta_i$ which spans $t_{\mathrm{ICAP},i} = t_{A,i} + t_{S,j} + t_{R,j} + t_{D,i}$ consecutive units of time (see Figure 3). During this time, the task is effectively set up in the FPGA. The fact of grouping together the task phases that need to access the ICAP is beneficial to schedule this limited resource in a more predictable way. However, note that $t_{\mathrm{ICAP},i}$ can vary depending on which task precedes $\theta_i$, and, therefore, it must be dynamically computed with each task release.

As previously introduced, R3TOS uses several ways to reduce $t_{\mathrm{ICAP},i}$ towards a higher performance. First, the direct access from a data consumer task to producer task's ODB results in $t_{D,i}$ and $t_{R,j}$ circumvention. Second, the use of DRTs to quickly relocate data between data buffers results in reduced $t_{D,i}$ and $t_{R,j}$. Finally, the reuse of previously configured tasks results in $t_{A,i}$ circumvention. In addition, $t_{E,i}$ can also be reduced by feeding the task with the highest clock rate. Furthermore, the configuration memory of the FPGA is used as a cache for both tasks and data. In fact, hardware tasks are deallocated from the FPGA only when their resources are required by other coming tasks, and the partial results computed by them are retrieved only when they are required by a software task.

The real-time constraint of R3TOS involves the existence of a relative *execution deadline* for each task, $D_i$, which is defined by the application programmer and represents the maximum acceptable delay for that task to finish its execution. The absolute execution deadline for each new task instance, $d_i$, is computed by adding the task release time, $r_i$, to its relative execution deadline, $d_i = D_i + r_i$. Even more important are the relative and absolute *set up deadline*s, $D_i^*$ and $d_i^*$, which represent the maximum acceptable delay for a task to start the computation in order to meet its execution deadline. A task is considered to be ready to start computing,

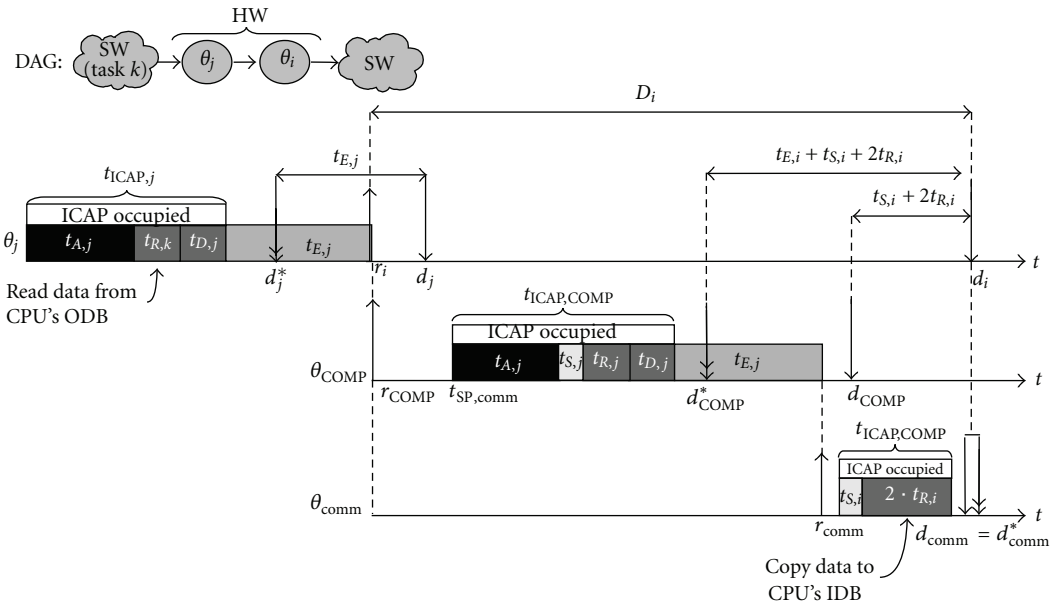



Figure 3: Consecutive execution of hardware tasks in R3TOS.

when it is completely configured in the device and the data to be processed is already loaded in its IDB. To achieve the predictability required by real-time behaviour, it is always considered the worst case that any of the aforementioned performance enhancements cannot be exploited.

As can be seen in Figure 3, the set up deadlines are different for hardware tasks which communicate with other hardware tasks and for those which deliver data to software tasks. This is because the data retrieval phase is included in the model of the data consumer hardware tasks, but it is not in the model of data consumer software tasks. Indeed, the data retrieval operation of a data consumer software task must be invoked from the data producer hardware task itself, which could interfere with the real-time behaviour.

The absolute set up deadline of a "standard task" which communicates with other hardware tasks, or which receives data from a software task, is equal to $d_i^* = d_i - t_{E,i}$. This is the case of $\theta_j$ in the aforementioned Figure 3. On the other hand, hardware tasks that deliver data to the main CPU, for example, $\theta_i$ in the figure, are modelled as two separate standard tasks in order to harmonize their management.

(i) An exclusively computing task $\theta_{\mathrm{COMP}}$ with $h_{x,\mathrm{COMP}} = h_{x,i}$, $h_{y,\mathrm{COMP}} = h_{y,i}$, $r_{\mathrm{COMP}} = r_i$, $t_{E,\mathrm{COMP}} = t_{E,i}$, and $t_{\mathrm{ICAP,COMP}} = t_{A,i} + t_{R,j} + t_{S,j} + t_{D,i}$, where $\theta_j$ is the data producer of $\theta_i$, $d_{\mathrm{COMP}} = d_i - t_{S,i} - 2 \cdot t_{R,i}$ and $d^*_{\mathrm{COMP}} = d_i^* - t_{E,i} - t_{S,i} - 2 \cdot t_{R,i}$. Note that the multiplication by two of $t_{R,i}$ accounts for the dual operation of reading data from task's ODB and copying it to CPU's IDB.

(ii) An exclusively communicating task $\theta_{\mathrm{comm}}$ with $h_{x,\mathrm{comm}} = h_{y,\mathrm{comm}} = 0$, $t_{\mathrm{ICAP,comm}} = t_{S,i} + 2 \cdot t_{R,i}$, $t_{E,\mathrm{comm}} = 0$, $r_{\mathrm{comm}} = t_{\mathrm{SP,COMP}} + t_{\mathrm{ICAP,COMP}} + t_{E,i}$, and $d_{\mathrm{comm}} = d^*_{\mathrm{comm}} = d_i$.

To achieve the predictability required by real-time behaviour, it is always considered the worst case execution time $t_{E,i}$ for the tasks. As a result, the HWS must be accessed only once, and, thus, $t_{S,i} = t_S$ for all $\theta_i$, where $t_S$ is equal to the time needed to read back a single frame from the FPGA device. Note that in this situation HWSs are checked only to confirm the correct ending of tasks' execution. With the same objective of achieving the highest predictability, the worst case $t_{D,i}$ and $t_{R,i}$ are considered as well, that is, the largest amount of data to exchange.

Without any loss of generality, it is assumed that there is no a priori knowledge of task release times, which, indeed, may depend on previous computations, that is, event-triggered data-dependant computing. Therefore, tasks are considered *sporadic* and *aperiodic*.

## 5. Real-Time Scheduling Algorithms

A nonpreemptive EDF algorithm suited to be used in the reconfigurable scenario we target in R3TOS is shown in Algorithm 1. This includes the capability to discard the tasks with greater area than available in the device early (see line 3), which is an easy but effective way to account for external fragmentation in the scheduler. This is the basis for the scheduling algorithm presented herein, the *Finishing Aware EDF* (FAEDF), which includes the capability to "look ahead" to find future releases of adjacent pieces of area, when executing tasks finish. This capability is designed to replace task preemption in the cases when preemption would be beneficial.

*5.1. Finishing Aware Earliest Deadline First (FAEDF).* When using FAEDF, a ready task $\theta_i$ which cannot be allocated on the FPGA at a given scheduling point $t_{\mathrm{SP}}$ is only discarded if there are not any sufficiently large tasks finishing before its deadline expires. If any, the allocation of $\theta_i$ is delayed until the executing task(s) finishes and there are enough free

```
    input: (a) List of R ready tasks, sorted by increasing d*_i − t_ICAP,i, and (b) MER, given by the
           allocator
    output: Scheduled Task
1   i ← First Task in Ready Queue sorted based on Time;
2   while i ≤ R do
3     if MER ≥ h_x,i · h_y,i then
4       if Allocate (i) ≠ ∅ then
5         return i;
6       end if
7     end if
8     i ← Next Task in Ready Queue;
9   end while
10  return ∅;
```

Algorithm 1: Schedule_EDF ().

resources in the device. The time left until then is used to schedule other ready tasks which can be completely set up before the deadline of $\theta_i$ expires. Note that despite these tasks that are also selected based on EDF, the scheduling policy is altered as tasks with farther deadlines can be scheduled before others with closer deadlines, but which do not meet the aforementioned set up time requirement. In case there are no ready tasks which meet the set up time requirement, FAEDF does not schedule any task; that is, it produces a "blank schedule," assuming the expected finish of the executing tasks is close enough.

The fact of wasting ICAP time when scheduling blank times may seem contradictory. However, this occurs only when the real-time constraints of the ready tasks are loose, that is, when $\sum_{\forall k \in \text{Ready Queue}}(t_{\text{ICAP},k}/d_k^* - t)$ is low, with $t$ referring to the actual time in the system. Otherwise, it is assumed that the fact of altering the EDF policy to give a "second chance" to a specific ready task to meet its deadline may lead to miss more deadlines, and, thus, it is discarded. That is, FAEDF proceeds as standard EDF when the real-time constraints of the ready tasks are tight, that is, when $\sum_{\forall k \in \text{Ready Queue}}(t_{\text{ICAP},k}/d_k^* - t)$ is high. Furthermore, note that during the scheduled blank times other R3TOS services which need to access the ICAP could be executed; for example, the configuration state of the HWuK could be checked in the configuration memory. Finally, not occupying the ICAP when the real-time constraints are loose allows to rapidly allocate any incoming task with very tight deadline.

The task set shown in Table 1 is used to illustrate the improvement brought about by FAEDF's "look ahead" capability with regard to nonpreemptive EDF. As shown in Figure 4(a), scheduling $\theta_3$ at $t_{\text{SP},B}$ delays the subsequent allocation of $\theta_2$ too long, and, as a result, the latter misses its deadline. Note that $\theta_2$ cannot be allocated at $t_{\text{SP},B}$ because there is no enough adjacent area in the FPGA. On the other hand, as shown in Figure 4(b), at $t_{\text{SP},B}$, FAEDF finds out that $\theta_2$ can be allocated later using the resources that will be released by $\theta_1$. The time until then is used to allocate $\theta_4$. After having allocated $\theta_4$, a blank schedule is produced because $\theta_3$ cannot be completely set up in the remaining time until $d_2^* - t_{\text{ICAP},2}$. Note that if $d_3^*$, which is met just in time in Figure 4(b), had been tighter, FAEDF would have functioned as standard EDF; that is, it would have scheduled $\theta_3$ at $t_{\text{SP},B}$, assuming that it is impossible to meet at the same time both $d_2^*$ and $d_3^*$.

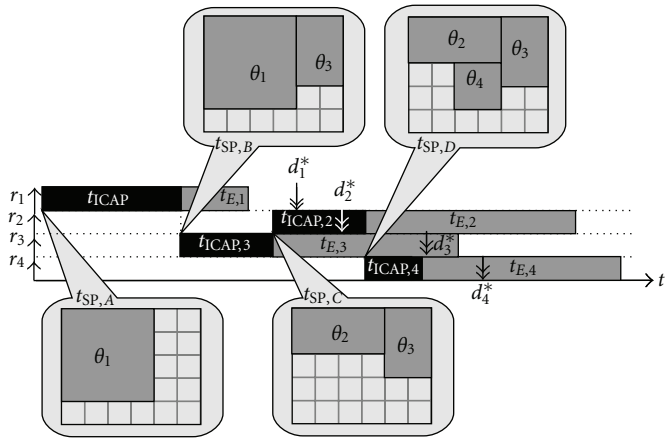
Table 1: Task-set used in Figure 4.

| | $h_{x,i}$ | $h_{y,i}$ | $t_{\text{ICAP},i}$ | $t_{E,i}$ | $D_i^*$ |
|---|---|---|---|---|---|
| $\theta_1$ | 4 | 4 | $6 \cdot t_{\text{KT}}$ | $3 \cdot t_{\text{KT}}$ | $11 \cdot t_{\text{KT}}$ |
| $\theta_2$ | 4 | 2 | $4 \cdot t_{\text{KT}}$ | $9 \cdot t_{\text{KT}}$ | $13 \cdot t_{\text{KT}}$ |
| $\theta_3$ | 2 | 3 | $4 \cdot t_{\text{KT}}$ | $8 \cdot t_{\text{KT}}$ | $17 \cdot t_{\text{KT}}$ |
| $\theta_4$ | 2 | 2 | $2 \cdot t_{\text{KT}}$ | $9 \cdot t_{\text{KT}}$ | $19 \cdot t_{\text{KT}}$ |

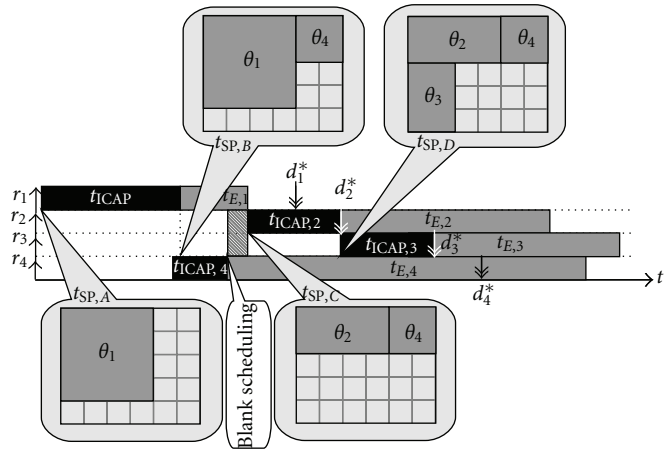
Algorithm 2 shows FAEDF's pseudocode. Lines 2 to 8 are exactly the same EDF algorithm presented in Algorithm 1, and lines 9 to 34 are the "looking ahead" extension to it. This extension is enabled only when the real-time constraints are slacker than a predefined *threshold* (see line 9). The executing queue is searched at lines 12 to 17 to find an executing task $\theta_j$ which would release sufficiently a large amount of resources when it finishes allocating the ready task $\theta_i$ ($h_{x,j} \geq h_{x,i}$ and $h_{y,j} \geq h_{y,i}$). If any, the ready queue is then scanned at lines 20 to 31 to find a ready task $\theta_m$ which can be completely allocated before the deadline of $\theta_i$ expires ($t_{\text{KT}} + t_{\text{ICAP},m} \leq d_i^* - t_{\text{ICAP},i}$). The scheduler produces a blank scheduling when it is possible to allocate $\theta_i$ using the resources to be released by an executing task $\theta_j$, but there are no $\theta_m$ ready tasks that can be completely set up before $d_i^*$ (line 31). Finally, with the objective of speeding up the execution of the scheduling algorithm, an array (`tried`) is used to keep track of the ready tasks that have been unsuccessfully tried to be placed (lines 4, 22, and 26).

The worst case complexity of FAEDF algorithm is $O(R \cdot (R + E))$ when the "look ahead" is enabled and $O(R)$ if it is not enabled. In any case, as previously explained, FAEDF includes several ways to effectively reduce the time overheads, for example, discard the ready tasks which are larger than the largest free rectangle on FPGA early or use the aforementioned `tried` array.

Finally, we note that our FAEDF algorithm does not consider the overheads introduced when making the scheduling




(a) Nonpreemptive EDF: $\theta_2$ misses its deadline




(b) FAEDF: Feasible scheduling

Figure 4: EDF and FAEDF scheduling.

and allocation decisions. These should be considered in a real-world application; for example, a guard increase of the ICAP access period of the tasks should be allowed.

## 6. Allocation Algorithms

As presented in Section 3, several research efforts can be found in the technical literature to improve the computation density when allocating hardware tasks onto an FPGA device. These approaches mainly include bin packing-based algorithms (e.g., [21, 22]) and adjacency-based heuristics (e.g., [17, 28, 29]). However, bin-packing algorithms do not consider the effect that placing a task into the "bin box" involves in allocating future tasks, and adjacency-based heuristics have a vision of only one resource row/column beyond the boundaries of already placed tasks. To tackle these limitations, the novel *Empty Area/Volume Compaction heuristics* (EAC and EVC) and, as a natural improvement to them, the *Snake* task allocation strategy are proposed in this article.

Before outlining these, it is important to note that all of the allocation algorithms presented in this section, including the ones we are proposing, manage the FPGA at a very high-level of abstraction. This is only possible because the FPGA area is kept void of static routes and other implementation-related obstacles at all times, as R3TOS is able to do. Indeed, the FPGA is modelled as a grid, named as `FPGA_state`, where each position represents an FPGA resource or a set of resources. Due to the existing reconfiguration granularity in current Xilinx partially reconfigurable FPGAs (see Section 2), all of the resources included within the same column of a clock region are mapped to the same position in the grid; that is, the vertical granularity must be a minimum number of clock regions. On the other hand, the horizontal granularity can be arbitrarily chosen based on the required efficiency in the use of system resources and admissible computational burden. The finest granularity and the best achievable exploitation of FPGA resources, that is, the exact number of resources required by the tasks, can be assigned to them, but with the highest computational burden.

*6.1. Empty Area/Volume Heuristics Compaction (EAC/EVC).* EAC and EVC heuristics are aimed at preserving the Maximal Empty Rectangle (MER) intact for future use as long as

```
input: (a) List of R ready tasks, sorted by increasing d*_i − t_ICAP,i
       (b) List of E executing tasks, sorted by increasing t_SP,i + t_ICAP,i + t_E,i
       (c) MER, given by the allocator
       (d) Real-time deadline tightness, Σ_{∀k∈Ready Queue} t_ICAP,k / (d*_k − t)
       (e) current time t_KT
output: Scheduled Task
1  Reset tried array (set all positions equal to false);
2  i ← First Task in Ready Queue;
3  while i ≤ R do
4    if MER ≥ h_x,i · h_y,i and tried[i] = false then
5      if Allocate(i) ≠ Ø then
6        return i;
7      end if
8    end if
9    if Σ_{∀k∈Ready Queue} t_ICAP,k / (d*_k − t) < Threshold then
10     j ← First Task in Executing Queue;
11     F ← false;
12     while j ≤ E and t_SP,j + t_ICAP,j + t_E,j ≤ d*_i − t_ICAP,i and !F do
13       if h_x,j ≥ h_x,i and h_y,j ≥ h_y,i then
14         F ← true;
15       end if
16       j ← Next Task in Executing Queue;
17     end while
18     if F then
19       m ← Next Task in Ready Queue after i;
20       while m ≤ R do
21         if t_KT + t_ICAP,m ≤ d*_i − t_ICAP,i then
22           if MER ≥ h_x,m · h_y,m and tried[m] = false then
23             if Allocate(m) ≠ Ø then
24               return m;
25             else
26               tried[m] ← true;
27             end if
28           end if
29         end if
30         m ← Next Task in Ready Queue;
31       end while
32       return Ø;
33     end if
34   end if
35   i ← Next Task in Ready Queue;
36 end while
37 return Ø;
```

Algorithm 2: Schedule_FAEDF().

possible, trying to place small tasks in the smallest pieces of empty area where they fit, including the areas between the damaged resources. As a result, these heuristics are suitable to be used in R3TOS. In fact, in the presence of faults it is not true that the best position to place a task is always in a vertex of a previously allocated task, as adjacency-based heuristics assume (e.g., 2DA or 3DA).

Another benefit of EAC and EVC heuristics is their ability to manage the FPGA device as a single resource instead of splitting it into nonrealistic independent pieces of area, as bin packing-based algorithms do (e.g., KAMER). Indeed, these heuristics thoroughly analyze the state of the whole reconfigurable area of the FPGA, giving rise to an Empty Area Descriptor (EAD), which is later consulted when a new task comes. By using the precomputed information included in the EAD, nonfeasible allocations can be early discarded and the quality of the feasible allocations can be determined very quickly. As a result, more placement candidates can be evaluated in less time, and, eventually, better results can be achieved.

The main difference between EAC and EVC is that the latter also analyzes the time domain to prevent future fragmentation in the device and to achieve higher computation densities.

Algorithms 3 to 7 show the most important pseudocode fragments to compute the EAC and EVC heuristics, and Algorithm 8 shows the pseudocode to make the allocation decisions based on these heuristics. Moreover, the example depicted in Figure 7 is used to illustrate the computation of these heuristics. All intermediate calculations related to this example are depicted in Figure 8.

*6.1.1. EAC/EVC: 1D Analysis.* The area of the FPGA is firstly analyzed in the horizontal direction, from right to left and from left to right (see Algorithm 3). Every time the resources associated to the cell in the next column of the `FPGA_state` grid are available to be used, a counter (named as `lenght` in Algorithm 3) is incremented (line 5), and in case the resources are not available, the counter is reset (lines 8 and 9). It is assumed that a resource is not available when it is already assigned to another executing task or when it is damaged, but it is considered available when assigned to a task that remains allocated but not performing active computation. Likewise, active data traces (i.e., those which are still required by other consumer tasks) are also represented by means of not available (BRAM) cells in the `FPGA_state` grid. Hence, the set of the counter values at each grid position makes up the Right/Left Adjacency Matrices (`RAM` and `LAM`) and represents the amount of adjacent free resources in each direction (right or left).

*6.1.2. EAC/EVC: 2D Analysis.* In a second phase, the 2D analysis is carried out (see Algorithm 4). The objective of this analysis is to find the greatest empty rectangle that can be formed at each position in up-right, up-left, down-right, and down-left directions and results in the four new matrices: Up-Right Adjacency Matrix (`URAM`), Up-Left Adjacency Matrix (`ULAM`), Down-Right Adjacency Matrix (`ULAM`), and Down-Left Adjacency Matrix (`DLAM`). These rectangles are depicted in Figure 5. The 2D analysis is based on the geometrical meaning of the `RAM`/`LAM` matrices. Indeed, the product of the `RAM`/`LAM` value stored in each position (named as `width` in Algorithm 4) multiplied by the number of consecutive neighbor locations, in up or down directions, with the same or greater value is equal to the area of the widest empty rectangle that can be formed at that position. Analogously, the area of the highest empty rectangle that can be formed at each position is equal to the product of the number of consecutive neighbor locations, in up or down directions, with a nonzero `RAM`/`LAM` value multiplied by the lowest among these values. In a general way, the area of all empty rectangles that can be formed at each position can be iteratively computed as the product of the successive decrements of the `RAM`/`LAM` value stored in that position (until 0) multiplied by the the number of consecutive neighbor locations, in up or down directions, with the same or greater `RAM`/`LAM` value. Note that this multiplication is computed by the repeated addition of the actual `width` value in line 9 of Algorithm 4. In this algorithm the value of `width` is decremented in line 15, and the $k$ index is used to go through up or down directions. The area of the greatest empty rectangle at each position and in each direction is finally written in the `URAM`, `ULAM`, `DRAM`, and `DLAM` matrices (lines 12, 13, and 19).

For instance, in Figure 8, the value of `URAM` (5, 9) corresponds to the area of $4 \times 7$ (widest) empty rectangle which can be formed in up-right direction. Note that this is the only rectangle that can be formed at that position. On the other hand, the value of `URAM` (5, 2) is equal to the area of $3 \times 4$ (highest) empty rectangle which can be formed in that direction. Note that at that position the widest rectangle that can be formed is $1 \times 7$. Finally, the value of `URAM` (7, 6) is equal to the area of either $5 \times 3$ (neither widest nor highest) empty rectangle or $3 \times 5$ (widest) empty rectangle. At that position the highest empty rectangle that can be formed is $7 \times 2$.

*6.1.3. EAC: Area Adjacency Analysis.* As shown in Algorithm 5, the third and last phase of EAC heuristic computation consists in adding the aforementioned four matrices `URAM`, `ULAM`, `DRAM`, and `DLAM` to give rise to the 2D Adjacency Matrix (`2DAM`). Conceptually, the values stored in each position of `2DAM` represent in what measure that position contributes to form adjacent pieces of empty area.

*6.1.4. EVC: Time and Area Adjacency Analysis.* As previously mentioned, EVC extends the area analysis to include the time domain. Although it is inspired by the 3DA heuristic, some changes are introduced with the objective of reducing the computational burden when making the allocation decisions (see Algorithm 6). In order to create a task-independent set of data which could be used at runtime for any coming task, a time window $T_W$ equal to the greatest execution time in the task set is chosen ($T_W = \max\{t_{E,i}\}$). For instance, in the example shown in Figure 7, $T_W = \max\{5, 8, 6\} = 8$.

For each position, the temporal adjacency within the time window with device's boundaries (lines 11, 20, 29, and 38), with other executing tasks (lines 8, 17, 26, and 37), and with damaged resources in the four directions (lines 6, 15, 24, and 33) is computed. As a result the Temporal Adjacency Matrix (`TAM`) is obtained. The value stored in each position of the `TAM` represents in what measure that position contributes to increasing the computation density. More specifically, a high temporal adjacency value means that the adjacent FPGA resources will remain occupied for a long time, while a low temporal adjacency value means that the adjacent resources will be released soon. For instance, in Figure 8, the top-left position has a temporal adjacency with device's boundaries equal to $8 + 0 + 0 + 8 = 16$, the same as for position (10, 2), but for the latter the adjacency is with a damaged resource and an executing task.

The temporal adjacency information (`TAM`) is then combined with the area adjacency information (`2DAM`) to create the 3D Adjacency Matrix (`3DAM`), as shown in Algorithm 7. The operation used to combine both time and area domains is the division (see line 3). Therefore, when the `2DAM` value is high (i.e., that location is part of a great adjacent free area) and the `TAM` value is low (i.e., that location does not contribute to keeping the tasks compacted), the resulting `3DAM` value is very high (i.e., disadvantageous); on the contrary, when the `2DAM` value is low and the `TAM` value is high, the resulting `3DAM` value

```
  input: FPGA_state (i.e. for each FPGA position Available or Not Available)
  output: RAM and LAM
1 for
  i = 0 ··· H_x − 1                                  /* when computing RAM or */
  i = H_x − 1 ··· 0                                     /* when computing LAM */
   do
2     lenght ← 1;
3     for j = 0 ··· H_y − 1 do
4         if FPGA_state[i][j] is Available then
5             lenght ← lenght + 1;
6             RAM[i][j]← lenght;                   /* when computing RAM or */
6             LAM[i][j]← lenght;                      /* when computing LAM */
7         else
8             lenght ← 1;
9             RAM[i][j]← 0;                        /* when computing RAM or */
9             LAM[i][j]← 0;                           /* when computing LAM */
10        end if
11    end for
12 end for
```

Algorithm 3: EAC heuristic: `Compute_RAM()` and `Compute_LAM()`.

is very low (i.e., advantageous). For the rest of the cases, the resulting `3DAM` value is medium.

*6.1.5. The Empty Area Descriptor (EAD).* As the tasks are placed relatively to their upper-left vertex, `DRAM` and `RAM` matrices are especially useful to describe the state of the FPGA reconfigurable area, being the central elements in the EAD. Each value stored in the `DRAM` matrix indicates the biggest empty rectangle available in the down-right direction. Therefore, a task can be placed in a given position only if its area is less than or equal to the actual `DRAM` value stored at that position. To account for shape aspects, the `RAM` matrix is used. A task can be placed in a given position only if its width is less than or equal to the actual `RAM` value stored at that position. In order to accelerate the search of feasible allocations for the tasks, the highest value in each column of the `DRAM` (named as `column_MER`) is also saved in the EAD. `column_MER`s permit to discard all the positions of a column without having to individually analyze each of these positions. The last element in the EAD is the MER, which is equal to the maximum value in the `DRAM`. As previously introduced, this value is given to the scheduler to discard unfeasible to place tasks early. The hierarchical structure of the EAD is shown in Figure 6.

*6.1.6. EAC/EVC-Based Allocation Decision Making.* At runtime, when a new task comes, the set of feasible allocations for it are evaluated based on the precomputed values stored in the `2DAM`, when using the EAC heuristic, or in the `3DAM`, when using the EVC heuristic. As shown in Algorithm 8, an EAC and an EVC score is assigned to each feasible allocation $(x, y)$ of a task $\theta_i$. Note that unfeasible allocations are discarded early based on the EAD (see lines 3 and 5). The EAC and EVC scores are computed as the sum of the `2DAM` or `3DAM` values corresponding to the resources to be assigned to $\theta_i$ in the allocation being evaluated (line 9). The placement quality is inversely proportional to the EAC and EVC scores. Conceptually, a low score means that the adjacent empty area in the device is not significantly fragmented when allocating the task in that position. In the case of EVC, a low score also ensures a good compactness of the tasks. Therefore, the final placement decision consists in selecting the feasible allocation with the lowest EAC or EVC score (lines 12, 13, and 14). Note that this way of functioning allows for dealing with different task shapes, that is, nonrectangular tasks.

The benefit of EAC and EVC when coping with permanent damage is illustrated in Figure 7. According to both 2DA and 3DA heuristics and considering the adjacency with the damage as well, $\theta_i$ would have been allocated at the bottom-left vertex of the FPGA (candidate B) with a 2DA score equal to 14 and a 3DA score equal to 74. The 2DA score for the candidate A is only 10, and the 3DA score for this candidate is only 56. On the other hand, the lowest EAC and EVC scores are obtained for candidate A; that is, EAC = 947 and EVC = 614. The scores for candidate B are EAC = $65 \cdot 20 = 1300$ and EVC = 623. Therefore, according to both EAC and EVC heuristics $\theta_i$ is placed at candidate allocation A. Hence, as shown in Figure 7, both 2DA and 3DA heuristics lead to a reduction of the MER from 48 (when placing the task at position A using either EAC or EVC) to 30, making it more difficult to allocate greater area tasks coming in the future.

Furthermore, by using the precomputed `2DAM` and `3DAM` matrices, the evaluation of each placement candidate for a task $\theta_i$ can be done very quickly, involving only $h_{x,i} \cdot h_{y,i}$ additions. Note that the required time for making the allocation decisions of large and small tasks tends to balance: while there are more feasible candidate allocations for a small task rather than for a large task, the quality of each candidate is evaluated faster for small tasks as the number of additions to be done is lower.

On the other hand, the EAD updating is a time-consuming process which is performed in parallel with the

```
   input: RAM (for URAM and DRAM computation) and
          LAM (for ULAM and DLAM computation)
   output: URAM, ULAM, DRAM and DLAM
1  for i = 0 ··· H_x − 1 do
2      for j = 0 ··· H_y − 1 do
3          width ←RAM[x][y];                    /* when computing URAM/DRAM or */
3          width ←LAM[x][y];                     /* when computing ULAM/DLAM */
4          area_max ← width;
5          k ← j;
6          while width > 0 do
7              area ← 0;
8              while k < H_y and k ≥ 0 and
                   RAM[i][k] ≥ width             /* when computing URAM/DRAM or */
                   LAM[i][k] ≥ width              /* when computing ULAM/DLAM */
               do
9                  area ← area + width;
10                 k ← k + 1;                    /* when computing URAM/ULAM or */
10                 k ← k − 1;                     /* when computing DRAM/DLAM */
11             end while
12             if area > area_max then
13                 area_max ← area;
14             end if
15             width ← width − 1;
16         end while
17     end for
18 end for
19 URAM[x][y]← area_max;                          /* when computing URAM or */
19 ULAM[x][y]← area_max;                          /* when computing ULAM or */
19 DRAM[x][y]← area_max;                          /* when computing DRAM or */
19 DLAM[x][y]← area_max;                           /* when computing DLAM */
```

Algorithm 4: EAC heuristic: Compute_URAM(), Compute_ULAM(), Compute_DRAM() and Compute_DLAM().

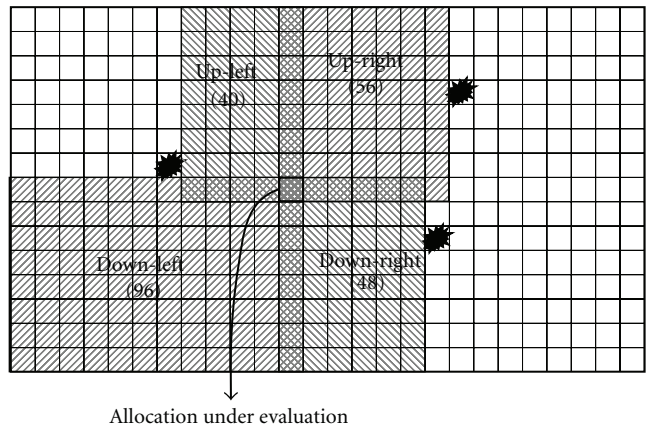



Figure 5: Greatest empty rectangles in up-right, up-left, down-right, and down-left directions.

setting-up of the last allocated task in order to improve system performance. Indeed, as the scheduling algorithm is not preemptive, the next scheduling point $t_{SP}$ will not be before the task is completely set up in the device. The 2DAM and 3DAM matrices are thus updated with the area state expected by then: the resources assigned to the task being set up are marked as not available, and the resources assigned to the executing tasks which are expected to finish by then are marked as available.

Overall, the worst case complexity of EAD updating is $O(3 \cdot H_x \cdot H_y + 4 \cdot H_x \cdot \sum_{i=1}^{i=H_y} i)$, whereas the runtime allocation decision making has a worst case complexity of $O(h_{x,i} \cdot h_{y,i} \cdot (H_x - h_{x,i} + 1) \cdot (H_y - h_{y,i} + 1))$. Unlike most of allocation algorithms, whose complexity depends on the number of allocated tasks, the complexity of EAC and EVC heuristics depends on the size of the FPGA. The benefit comes from the fact that not feasible candidates can be discarded early

```
  input: URAM, ULAM, DRAM and DLAM
  output: 2DAM
1 for i = 0 ··· H_x − 1 do
2    for j = 0 ··· H_y − 1 do
3        2DAM←ULAM[i][j] + URAM[i][j] + DLAM[i][j] + DRAM[i][j];
4    end for
5 end for
```

Algorithm 5: EAC heuristic: `Compute_2DAM()`.

by consulting the EAD, significantly reducing the effective amount of time needed to make the allocation decisions.

With the amount of time available to update the EAD limited by the shortest ICAP access period of the tasks, that is, $\min\{t_{\text{ICAP},i}\}$, an effective way to speed up this process is to increase the used granularity at the expenses of losing efficiency in the management of FPGA resources. Note that efficiency is of outmost importance when using small FPGAs, where EAD updating time is not so critical, but it is less important when using large FPGAs which involve longer updating times. Based on this and also arguing that the type of computation necessary to update the EAD is suitable to be accelerated by hardware (e.g., Algorithms 3 to 7 have regular data dependencies, and, as shown in Figure 8, the `URAM`, `ULAM`, `DRAM`, and `DLAM` matrices can be concurrently computed), we posit that the amount of time needed to complete the updating can be kept within reasonable bounds, enabling the use of the proposed heuristics in future FPGA devices, with presumably faster reconfiguration speed and more logic resources. A hardware implementation of an EAD updater is described in Section 7.

*6.2. Snake Task Allocation Strategy.* While EAC and EVC reduce at maximum the negative effect provoked by external fragmentation, they do not directly consider some key aspects in RC, such as intertask communications (i.e., hardware tasks are assumed to be independent), usable clock frequency, and FPGA resource heterogeneity. Hence, the allocation decisions may result in low performance due to intensive use of ICAP to exchange data among tasks or due to the fact that tasks are not executed at their highest clock rate. To tackle these issues the *Snake* allocation strategy is proposed.

Besides promoting the reuse of previously configured circuitry, *Snake* also tries to reuse intermediate partial results between different computation stages when dealing with noncritical HBC tasks. Note that when a task is noncritical a single instance of it is executed on the FPGA and there is no need to check the correctness of its results by accessing them through the ICAP. On the other hand, EAC and EVC heuristics continue to be useful for allocating redundant critical tasks and noncritical LBC tasks. That is, noncritical HBC tasks (which need long time to exchange data through the ICAP) are allocated with the objective of reducing the ICAP occupation, at the expense of increasing external fragmentation on the device, and LBC tasks (which need short time to exchange data through the ICAP) are efficiently allocated on the resulting FPGA, area fragments.

With the objective of speeding up the computation, *Snake* tries to execute each task at its highest allowed clock frequency, especially LBC tasks. However, this must be carefully treated to avoid allocation problems due to the fact that each FPGA clock region can allocate a maximum of two tasks running at different clock frequencies; that is, there are only two regional clock nets to distribute the clock signals in a clock region. In order to deal with this limitation *Snake* promotes the allocation of tasks with similar clock rates together in the same or adjacent rows, while it tries to allocate the tasks with radically different clock frequencies in separate rows. This permits to make up large regions with the same clock domain where future (large) tasks could be allocated.

Summing up, while reusing circuitry and partial results speeds up the set up phase of the tasks (better use of ICAP: time), an optimal management of clocking resources accelerates their execution phase (better use of FPGA resources: area). However, usually it is impossible to simultaneously take advantage of both improvements. When the execution time of a task is significantly longer than its set up time (i.e., LBC tasks), it is preferable to feed the task with the highest clock rate although this results in longer ICAP occupation. On the other hand, when the set up time of a task is in the same range of its execution time (i.e., HBC tasks), circuitry and/or data reuse is promoted. Indeed, note that HBC tasks usually complete their computation within a relatively short amount of time, and hence the occupation of the clocking resources is not a major problem.

In order to increase the allocatability of the tasks that include more scarce BRAM-based data buffers (typically HBC tasks), several versions of the same task are provided. As shown in Figure 9, each of the task versions uses different IDB and ODB locations and defines a different direction of the computation; that is, data flow from the IDB to the ODB. By using the appropriate task version at each time, the task can leave its results in the easiest accessible BRAM memories to be accessed by the subsequent data consumer tasks. In Figure 9, the white arrows represent the computation direction, which is vertical for task versions from *a* to *d* and horizontal for versions from *e* to *i*. Aiming at best exploiting the FPGA resources, the tasks must always span a minimum number of clock regions in height, and, given the granularity of FPGA's configuration memory, the size of the IDB and

```
    input: FPGA_state and the state of the tasks
    output: TAM
1   for i = 0 ··· H_x − 1 do
2     for j = 0 ··· H_y − 1 do
3       TAM[i][j] ← 1;
        // Bottom
4       if j − 1 ≥ 0 then
5         if FPGA_state[i][j − 1] is Damaged then
6           TAM[i][j] ←TAM[i][j] + T_W;
7         else if Task @ (i, j − 1) is Executing then
8           TAM[i][j] ←TAM[i][j] + Remaining t_e of Task @ (i, j − 1);
9         end if
10      else
11        TAM[i][j] ←TAM[i][j]+T_W;
12      end if
        // Left
13      if i − 1 ≥ 0 then
14        if FPGA_state[i − 1][j] is Damaged then
15          TAM[i][j] ←TAM[i][j] + T_W;
16        else if Task @ (i − 1, j) is Executing then
17          TAM[i][j] ←TAM[i][j] + Remaining t_e of Task @ (i − 1, j);
18        end if
19      else
20        TAM[i][j] ←TAM[i][j] + T_W;
21      end if
        // Top
22      if j + 1 < H_y then
23        if FPGA_state[i][j + 1] is Damaged then
24          TAM[i][j] ←TAM[i][j] + T_W;
25        else if Task @ (i, j + 1) is Executing then
26          TAM[i][j] ←TAM[i][j] + Remaining t_e of Task @ (i, j + 1);
27        end if
28      else
29        TAM[i][j] ←TAM[i][j] + T_W;
30      end if
        // Right
31      if i + 1 < H_x then
32        if FPGA_state[i + 1][j] is Damaged then
33          TAM[i][j] ←TAM[i][j] + T_W;
34        else if Task @ (i + 1, j) is Executing then
35          TAM[i][j] ←TAM[i][j] + Remaining t_e of Task @ (i + 1, j);
36        end if
37      else
38        TAM[i][j] ←TAM[i][j] + T_W;
39      end if
40    end for
41  end for
```

Algorithm 6: EVC heuristic: `Compute_TAM()`.

ODB should be an integer multiple of 4 BRAMs (72 Kb). However, for efficiency reasons, note that both IDB and ODB could be mapped to the same BRAM column, that is, each buffer using 2 BRAMs. In the horizontal direction the criterion changes. Tasks with horizontal computation direction must lie between two columns of BRAMs, and the width of tasks with vertical computation direction is chosen with the only constraint of fitting the necessary amount of resources. This means that a pair *a-c*, *a-d*, *b-c*, or *b-d* task versions can flexibly exploit the FPGA resources between two BRAM columns. We acknowledge a memory requirement increase to store the bitstreams associated to each version of the tasks. However, the memory overhead is admissible considering the benefit this method allows.

Figure 10 shows the allocation decision-making diagram of *Snake*. When a task $\theta_i$ is scheduled, *Snake* checks whether it is critical or noncritical and HBC or LBC. If $\theta_i$ is noncritical and HBC, *Snake* checks whether it is preferable to reuse circuitry (i.e., circumvent $t_{A,i}$) or reuse partial results (i.e., circumvent $t_{R,j}$ and $t_{D,i}$, where $\theta_j$ is the data producer task

```
   input: URAM, ULAM, DRAM, DLAM and TAM
   output: 3DAM
1  for i = 0 ··· H_x − 1 do
2     for j = 0 ··· H_y − 1 do
3        3DAM←2DAM[i][j]/TAM[i][j];
4     end for
5  end for
```

Algorithm 7: EVC heuristic: `Compute_3DAM()`.

```
   input: 2DAM (when using EAC), 3DAM (when using EVC) and θ_i
   output: Allocation (x,y)
1  EAC_min ← 4 · H_x · H_y;                                  /* when using EAC or */
1  EVC_min ← 4 · T_W · H_x · H_y;                            /* when using EVC */
2  for i = 0 ··· H_x − 1 do
3     if column_MER[i] ≥ h_x,i · h_y,i then
4        for j = 0 ··· H_y − 1 do
5           if DRAM[i][j] ≥ h_x,i · h_y,i and RAM[i][j] ≥ h_x,i then
6              EAC ← 0;                                     /* when using EAC or */
6              EVC ← 0;                                     /* when using EVC */
7              for m = i ··· i + h_x,i − 1 do
8                 for n = j ··· j + h_y,i − 1 do
9                    EAC ← EAC + 2DAM[m][n];               /* when using EAC or */
9                    EVC ← EVC + 3DAM[m][n];               /* when using EVC */
10                end for
11             end for
12             if EAC < EAC_min                             /* when using EAC or */
                  EVC < EVC_min                             /* when using EVC */
               then
13                x ← i;
14                y ← j;
15             end if
16          end if
17       end for
18    end if
19 end for
20 if
      EAC_min ≠ 4 · H_x · H_y                               /* when using EAC or */
      EVC_min ≠ 4 · T_W · H_x · H_y                         /* when using EVC */
   then
21    return (x, y);
22    else
23    return Ø;
24 end if
```

Algorithm 8: Allocation selection based on the EAC and EVC heuristics: `Allocate_EAC()` and `Allocate_EVC()`.

of $\theta_i$). Note that while circuitry can be reused only if the task remains still configured on the FPGA, partial results can always be potentially reused as the data traces of HBC tasks are stored in BRAMs. Reusing an already configured task is immediate, and no allocation decisions must be made; that is, the task is simply executed on the same position where it was last time. However, when reusing the partial results, some allocation decisions are to be made. The best case is when $\theta_i$ can directly access its input data from $\theta_j$'s ODB (i.e., $\theta_j$'s ODB is used as $\theta_i$'s IDB), but this requires there are sufficiently large amount of contiguous resources to allocate $\theta_i$ next to $\theta_j$'s ODB. Preferably, consumer tasks are allocated opposite to producer tasks, to keep the latter allocated on the FPGA, promoting future circuitry reuse. However, if strictly necessary producer tasks are deallocated and their resources assigned to consumer tasks. If there are several versions of $\theta_i$ which fit in the free area next to $\theta_j$'s ODB, EAC/EVC heuristics are used to select the one that minimizes

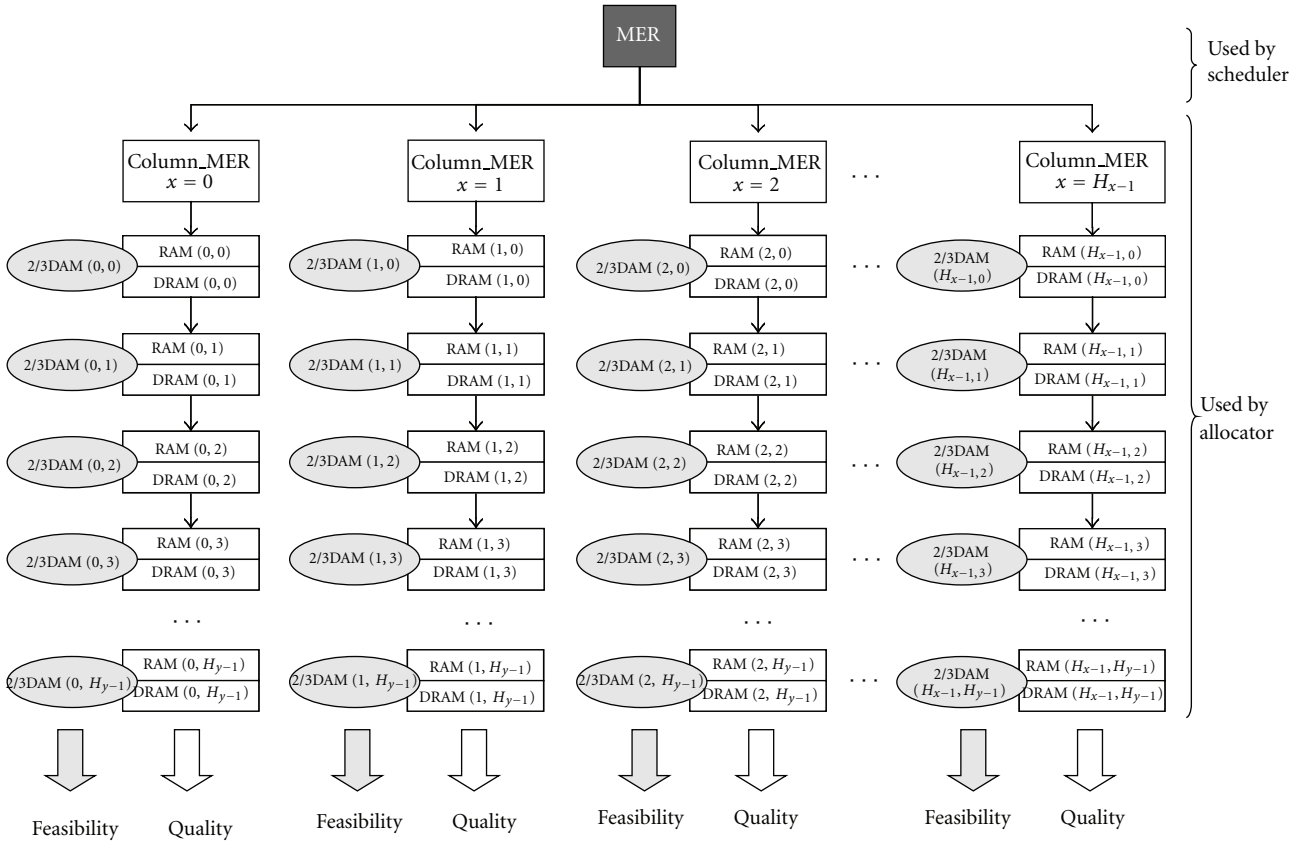



Figure 6: EAD structure.

the fragmentation on the device. On the other hand, if there is no sufficiently large free area to allocate any implementation version of $\theta_i$ next to $\theta_j$'s ODB, the feasibility of using a Data Relocation Task (DRT) is evaluated. By using a DRT the set of data can be rapidly moved (through the functional layer) from its current location to a new position where it is accessible by the consumer task. If no DRT can be used, the allocation decisions are made using the EAC/EVC heuristics, and the data is delivered to the consumer task through the configuration layer. Summing up, for noncritical HBC tasks, *Snake* starts evaluating the feasible allocations near the data producer task and continues evaluating the FPGA allocations which are reachable by means of DRTs, and finally it switches to analyze the whole FPGA locations seeking for the lowest EAC/EVC score, that is, minimal fragmentation.

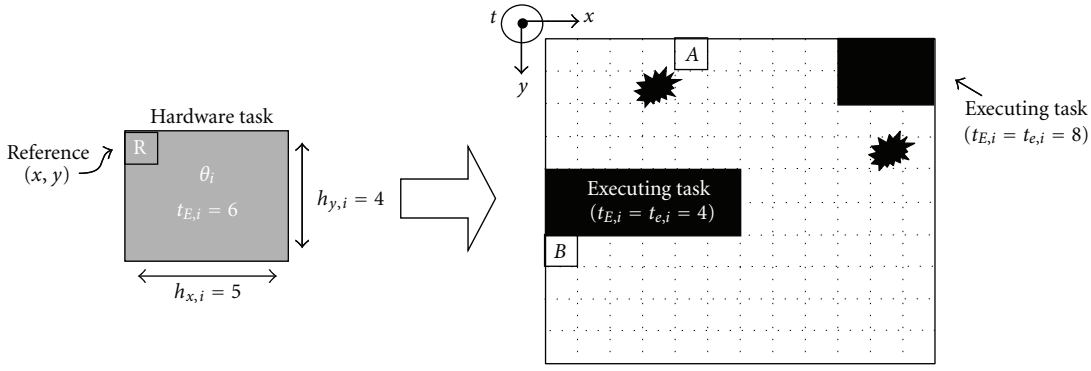
As shown in Figure 11, the linking together of the hardware tasks by means of the memory elements where the data traces are temporarily stored leads to computation chains on the FPGA. Indeed, this gives *Snake* its name. The task chains are initiated in the main CPU's ODBs (*Heads*), and the results computed by the last task in the chain are copied through the configuration layer of the FPGA to the main CPU's IDBs (*Tails*), where they are accessible by the software program. As shown in Figure 11(b), *Snake* is an efficacious way to deal with the heterogenous resource columns embedded in modern FPGAs as well as to circumvent the damaged resources in the chip.

## 7. Simulation Results

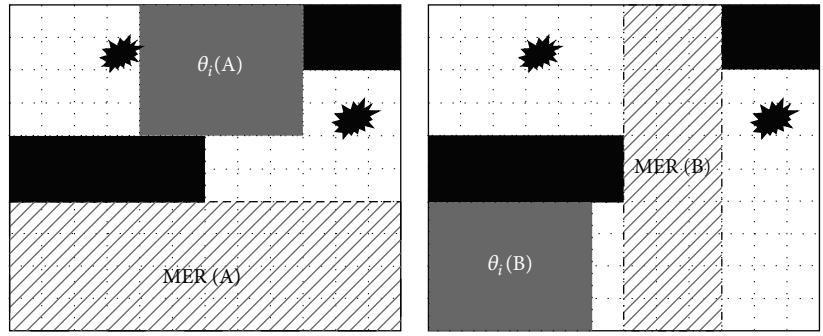
This section presents the obtained results when simulating our scheduling and allocation algorithms. The simulation experiments cover a wide range of task parameters and different damage situations in the chip. Finally, an estimation of the performance improvement brought about by *Snake* using a realistic heterogeneous FPGA device model is provided.

*7.1. Simulation Set up .* A discrete-time simulation framework was built to evaluate the performance of the proposed scheduling and allocating algorithms. The framework ran under Windows XP OS on an Intel Core Duo CPU @ 3 GHz.

The framework simulated a Virtex-4 XC4VLX160 device with up to 12 clock regions, 3 BRAM columns, 1 DSP48 column, and a sandbox of 28 CLB columns width. Based on the layout of this FPGA, shown in Figure 12(a), the vertical granularity was set to be a clock region, that is, $H_y = 12$, while the horizontal granularity was set to be either 4 CLB columns or a single heterogeneous resource column, that is, $H_x = 15$.




(a) FPGA area state ($t_{e,i}$ is the remaining execution time of task $i$)




(b) Candidate A ($MER_A = 48$) (c) Candidate B ($MER_B = 30$)

FIGURE 7: Allocating $\theta_i$ at candidate positions A and B.

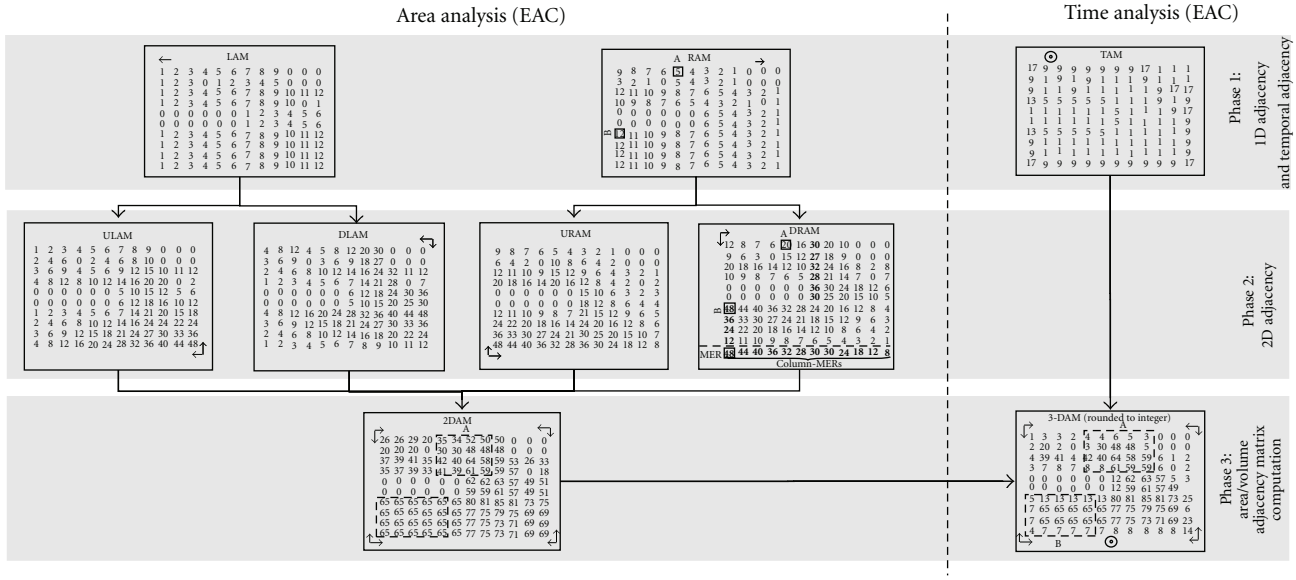



FIGURE 8: EAC and EVC heuristics computation.

Due to the lack of a common benchmark for RC systems, we resorted to creating our own synthetic hardware tasks. Different task sets, each containing up to 60 hardware tasks, were randomly generated. The execution deadlines, execution times, and sizes of the tasks were appropriately chosen, starting from random values, in order to simulate different $U_{\text{ICAP}}$ and $U_{\text{COMP}}$ situations. For simplicity, the allocation time of the tasks was considered to be equal to their size $t_{A,i} = h_{x,i} \cdot h_{y,i}$. We also considered that successive instances of the tasks were released with the shortest allowed time between them.

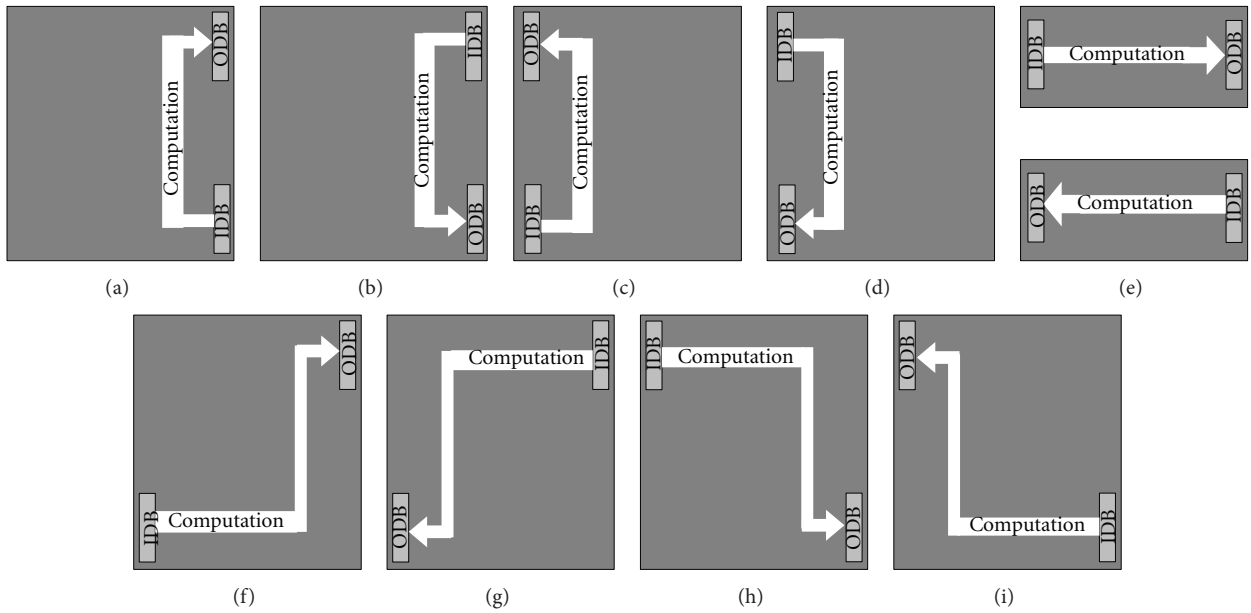



Figure 9: Different implementation versions for the hardware tasks.

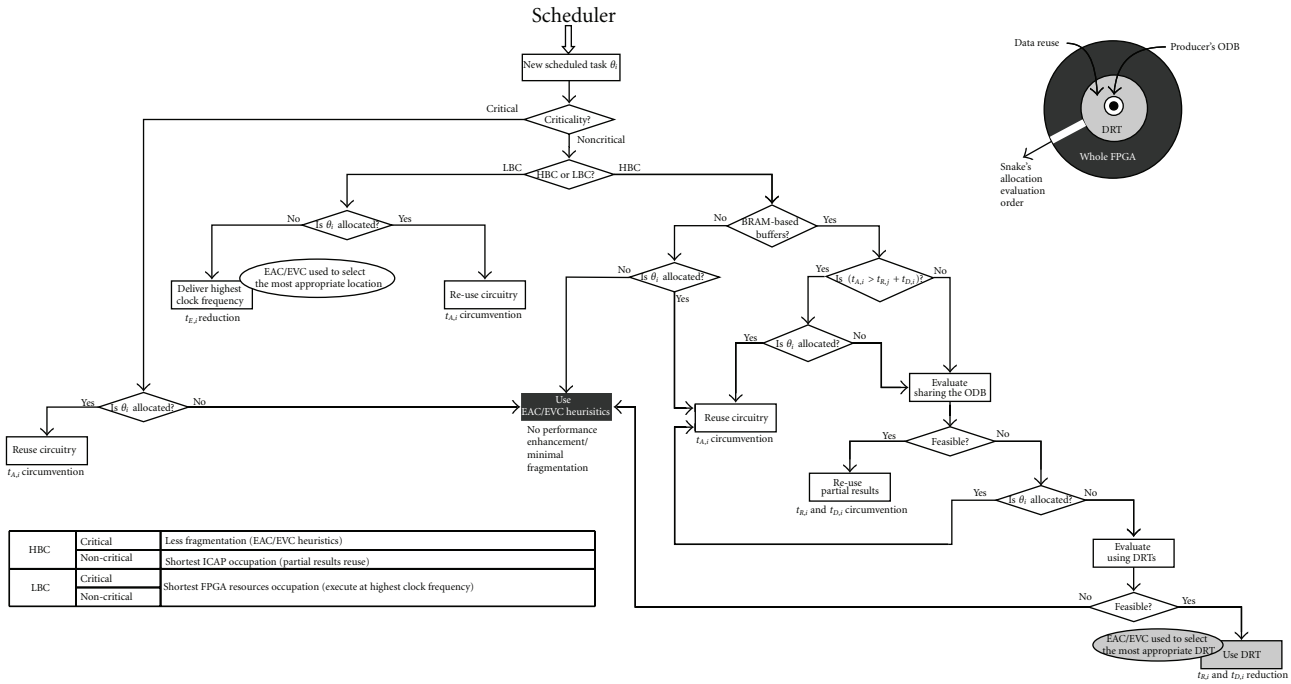
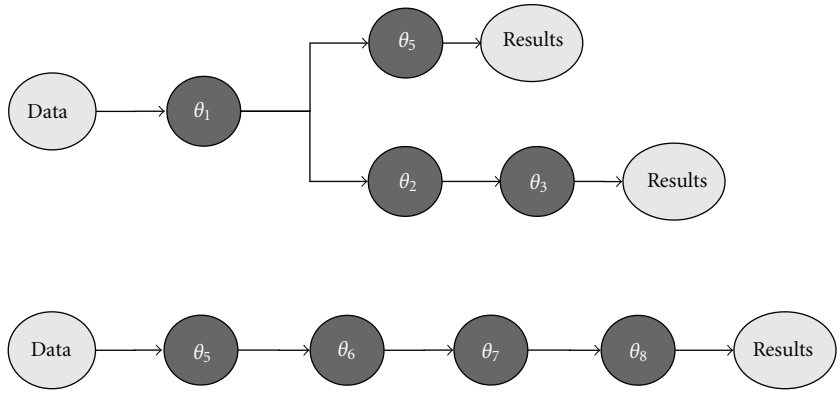


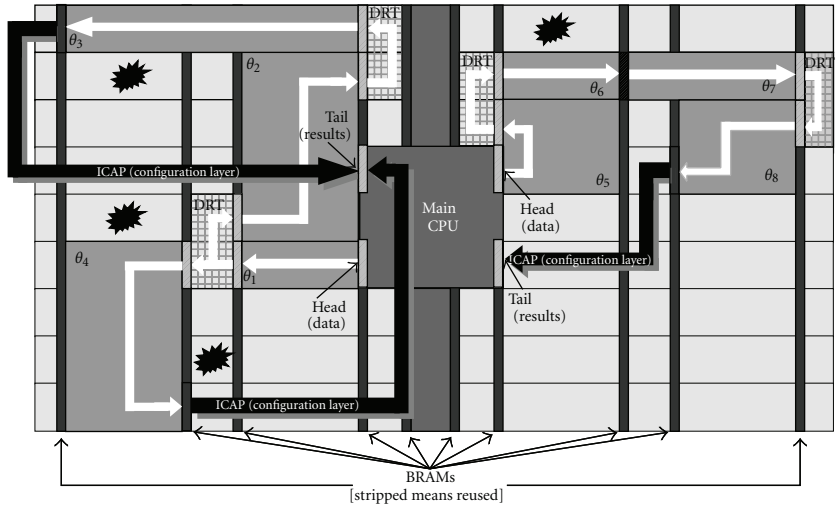

Figure 10: *Snake* task allocation strategy.

Due to the lack of a common benchmark for RC systems, we resorted to creating our own synthetic hardware tasks. Different task sets, each containing up to 60 hardware tasks, were randomly generated. We considered that successive instances of the tasks were released with the shortest allowed time between them. Note that with this worst case assumption, aperiodic tasks can be considered periodic. The execution deadlines, execution times, and sizes of the tasks were appropriately chosen, starting from random values, in order to simulate different real-time constraints and FPGA resource requirements. These are represented with $U_{\text{ICAP}}$ and $U_{\text{COMP}}$ parameters, where $U_{\text{ICAP}} = \sum_{\forall\theta_i}(t_{\text{ICAP},i}/D_i^*)$ and $U_{\text{COMP}} = (1/H_x \cdot H_y) \cdot \sum_{\forall\theta_i}((t_{\text{ICAP},i} + t_{E,i}) \cdot h_{x,i} \cdot h_{y,i}/D_i)$. For simplicity, the allocation time of the tasks was considered to be equal to their size $t_{A,i} = h_{x,i} \cdot h_{y,i}$.

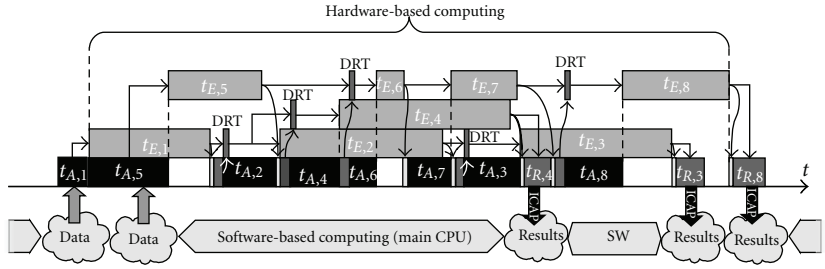
Up to 10,000 experiments were performed for each $U_{\text{ICAP}}$ and $U_{\text{COMP}}$ situations, and the obtained results were averaged. All tasks were set ready at time 0 (i.e., critical




(a) Reconfigurable application's DAG




(b) Allocation graph and general functioning scheme

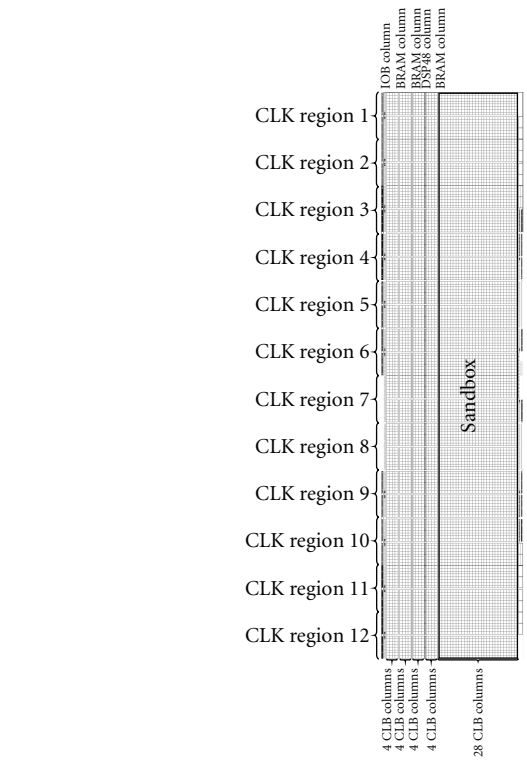



(c) Time diagram

Figure 11: Reusing partial results with Snake.

instant), and each experiment was considered to be finished when every task in the task set had either met or missed its execution deadline at least once.

In a first instance, we did not consider intertask dependencies, intertask communication overheads, or resource heterogeneity, and we focussed exclusively on the sandbox of the simulated FPGA. Furthermore, all tasks were considered to run at the same clock frequency. We note that this is the most commonly simulated scenario in related work. Since our scheduling and allocation algorithms are soft real-time, nonpreemptive and designed for 2D area model, they were only compared with equivalent nonpreemptive EDF scheduling, working with 2DA/3DA allocation heuristics. Indeed, EDF is one of the most consolidated soft real-time scheduling algorithms, and adjacency-based heuristics show the best allocation results in the current state-of-the-art, being used or serving as inspiration, in some of the latest research efforts in the field (see Section 3). In order to complete the characterisation of our EAC/EVC heuristics, up to 25 CLBs within the sandbox were marked as damaged (approximately 0.5% of the total CLBs in the sandbox). For fair comparison, we provided 2DA and 3DA heuristics with a




(a) FPGA layout

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CLB | BRAM | CLB | BRAM | CLB | DSP48 | CLB | BRAM | CLB | CLB | CLB | CLB | CLB | CLB | CLB |

(b) FPGA descriptor

Figure 12: Left half part of the simulated XC4VLX160 part.

mechanism for dealing with faults namely, the damaged CLB positions were added as additional vertexes in the Virtex List Set (VLS).

Four metrics were used to evaluate the performance of our algorithms.

(1) *Missed Deadlines* (MD) is the percentage of missed execution deadlines.

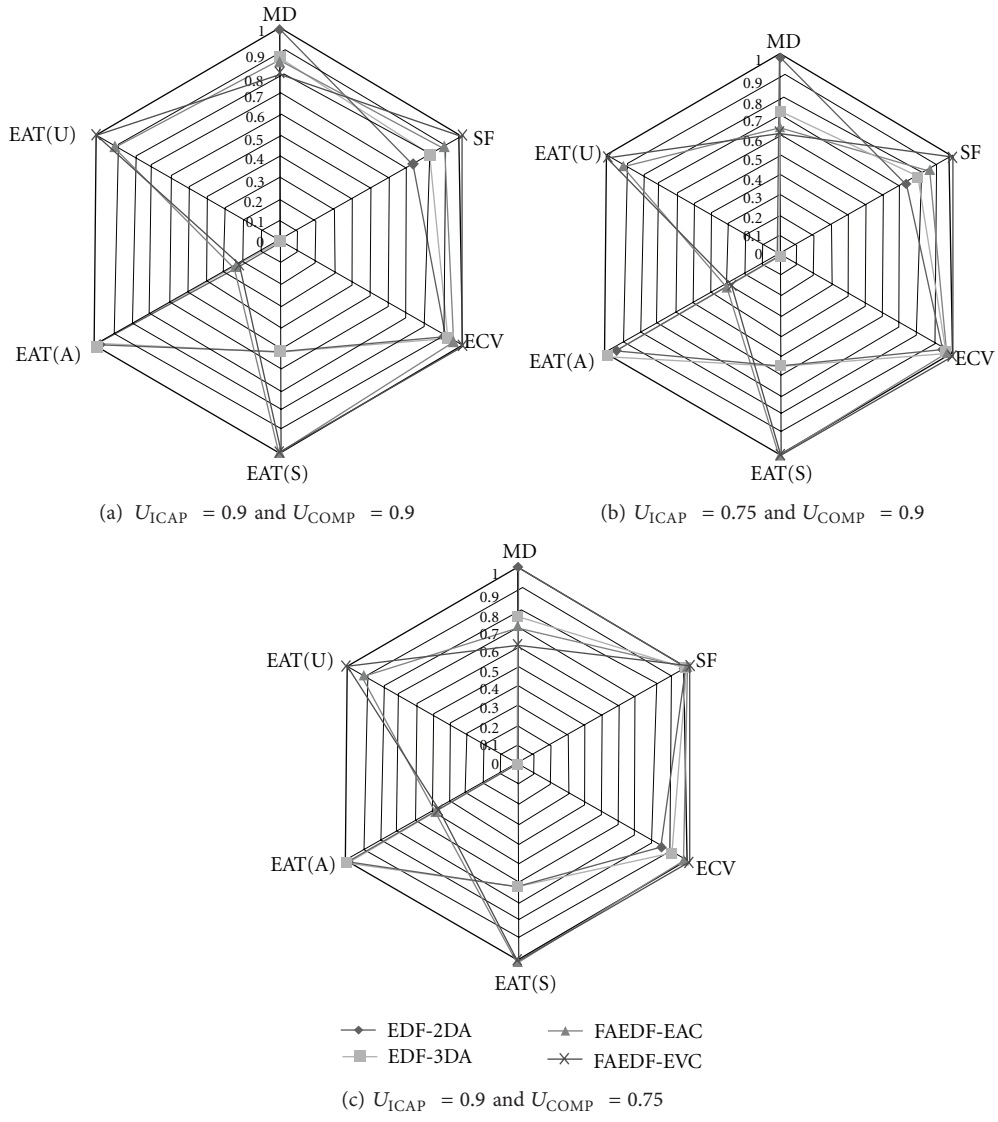
(2) *Scheduling Feasibility* (SF) refers to the percentage of feasible schedules produced, that is, percentage of schedules that do not miss any deadline.

(3) *Exploited Computation Volume* (ECV) refers to the use of the 3D computing space delivered by the FPGA (i.e., area time) to execute hardware tasks which meet their deadlines. Note that the set up phase of the tasks is not considered.

(4) *Algorithm's Execution Time* (AET) refers to the amount of time needed for making the scheduling and allocation decisions per executed task, as well as the time needed for updating the EAD.

In a second instance, all of the previously neglected RC-related issues were included in the simulation to evaluate our realistic *Snake* task allocation strategy. Hence, this simulation considered the whole FPGA device, that is, sandbox and heterogeneous resource columns. Intertask dependencies were randomly generated, with a maximum of 3 dependencies per task, and the amount of time needed to exchange data among tasks was also considered, that is, $t_{D,i}$ and $t_{R,i}$. The data delivery/retrieval time was uniformly distributed in $(90\% \cdots 110\%)$ of the execution time for HBC tasks and in $(30\% \cdots 50\%)$ for LBC tasks. HBC tasks and LBC tasks were randomly generated with a similar proportion of BRAM to CLB columns in the device, that is, 15 to 1. It was assumed that four implementation versions were available for each task, with vertical and horizontal computation direction, and for each computation direction with the IDB and ODB located in reverse positions. The data buffers of HBC tasks were considered to be implemented using 4 BRAMs. When DRTs could be used, it was assumed an acceleration factor of 1.5x in intertask communications. Furthermore, clocking aspects were envisaged: the execution time of the tasks depended on the used clock frequency, and the amount of tasks running at different clock frequencies in a row was limited to two. The highest clock frequency at which each task could run was randomly selected, ranging from 1x (i.e., base clock rate) to 5x.

*7.2. No Damage in the Device.* Figure 13 shows the collected results in three representative situations with no damaged in the simulated FPGA device: (a) when the FPGA resources are




(a) $U_{\text{ICAP}} = 0.9$ and $U_{\text{COMP}} = 0.9$

(b) $U_{\text{ICAP}} = 0.75$ and $U_{\text{COMP}} = 0.9$

(c) $U_{\text{ICAP}} = 0.9$ and $U_{\text{COMP}} = 0.75$

Figure 13: Performance with no damage in the FPGA device. EAT(U) refers to the EAD updating time, EAT(A) refers to the time needed for making the allocation decisions, and EAT(S) is the time needed for making the scheduling decisions.

highly utilized ($U_{\text{COMP}} = 0.9$) and the real-time constraints are tight ($U_{\text{ICAP}} = 0.9$), (b) when the FPGA resources are highly utilized and the real-time constraints are moderate ($U_{\text{ICAP}} = 0.75$), and (c) when the FPGA utilization is medium ($U_{\text{COMP}} = 0.75$) and the real-time constraints are tight. Note that the results shown in this figure are normalized to the highest value.

As expected, the results are better when either time aspects were considered when making the allocation decisions or when area aspects were considered when making the scheduling decisions; that is, EVC outperforms EAC, and FAEDF outperforms EDF. The improvement is more noticeable when the extra dimension was considered only once; that is, the benefit of including time aspects when making the allocation decisions is greater with EDF, which does not account for area aspects, than with FAEDF, which already considers area aspects. Specifically, in all of the simulated situations FAEDF-EVC shows the best results, that is, less amount of missed deadlines, higher rate of feasible schedules, and better exploitation of computation volume, while EDF-2DA shows the worst results. Moreover, the experiments conducted confirm that FAEDF-EAC produces slightly better results than EDF-3DA.

Including area aspects when making scheduling decisions FAEDF results in approximately double execution time, while EAC and EVC heuristics result in time overheads for updating the EAD. While the penalty for using FAEDF is admissible; that is, the scheduling decisions can always be made in less than 8 microseconds, the average time needed for updating the EAD is about 100 microseconds. Although this time might seem excessive, it is important to note that the conducted simulation does not account for the acceleration brought about by custom hardware implementation and parallelism in the area matrices computation. In contrast

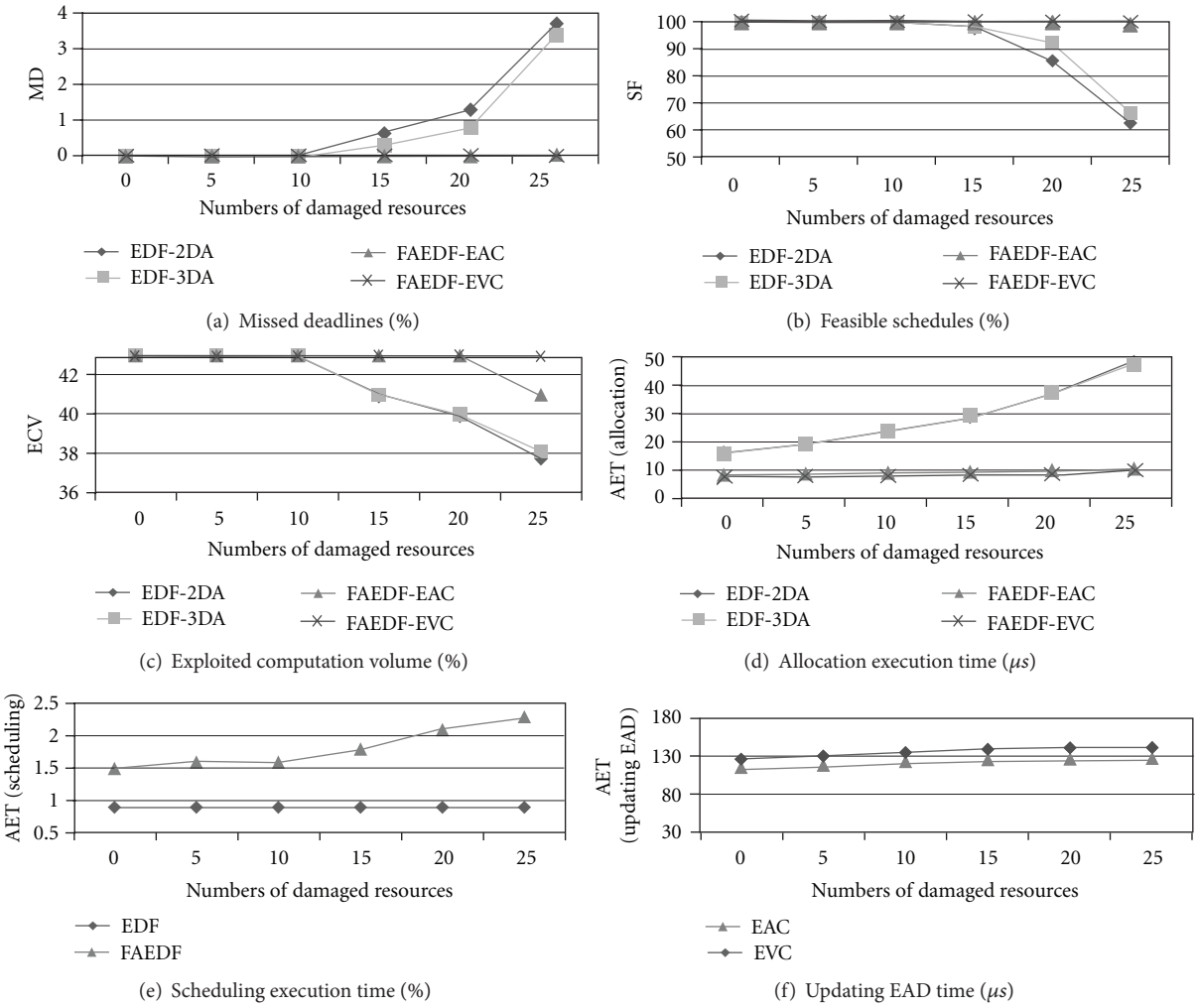



Figure 14: Low utilization of the FPGA and loose real-time constraints: $U_{\text{ICAP}} = 0.5$ and $U_{\text{COMP}} = 0.5$.

with the long time required to update the EAD, we note the remarkable achievable acceleration for making the allocation decisions when using our allocation heuristics. Indeed, the average time elapsed for making the allocation decisions is around 35 microseconds when using 2DA/3DA heuristics, and it is less than 10 microseconds when using our EAC/EVC heuristics (around 25% speed-up).

Summing up, when using FAEDF-EAC/EVC, the scheduling and allocation decisions can be made online in less than 20 microseconds (this time is approximately doubled when using EDF-2DA/3DA), but there is an overhead due to EAD updating process, which is in the range of 100 microseconds in our simulation framework. Therefore, the efficacy of our algorithms highly depends on the success in speeding up the EAD updating process. Note that some overhead is still admissible as the EAD updating can be parallelized with the task set up phase.

*7.3. Damage in the Device.* Figures 14 to 17 show the measured performance for various FPGA utilization situations and different real-time constraints in the presence of permanent damage on the FPGA device. As can be seen in the figures, most of the performance metrics (e.g., MD, SF, and EVC) show an exponential variation with the number of simulated faults in the FPGA.

The most important conclusion obtained from these results is the capability of EAC/EVC heuristics to deal with permanent damage in the FPGA. For instance, unlike 2DA/3DA, EAC/EVC heuristics are able to produce feasible schedules (i.e., no missed deadlines) for all of the simulated fault situations when the FPGA utilization is low ($U_{\text{COMP}} = 0.5$, see Figures 14 and 16). Although it is not possible to produce feasible schedules in the rest of the cases, the differences between the results obtained by both heuristics are still appreciable. Namely, when using EAC/EVC, between 5% and 20%, fewer deadlines are missed, and a similar improvement is measured in the exploitation of FPGA's computation volume. The difference is greater when the FPGA is highly utilized, that is, 20% improvement when $U_{\text{COMP}} = 0.9$ (see Figure 17), while 5% improvement when $U_{\text{COMP}} = 0.75$ (see Figure 15). In addition, it is interesting to check that EAC/EVC heuristics show better results with

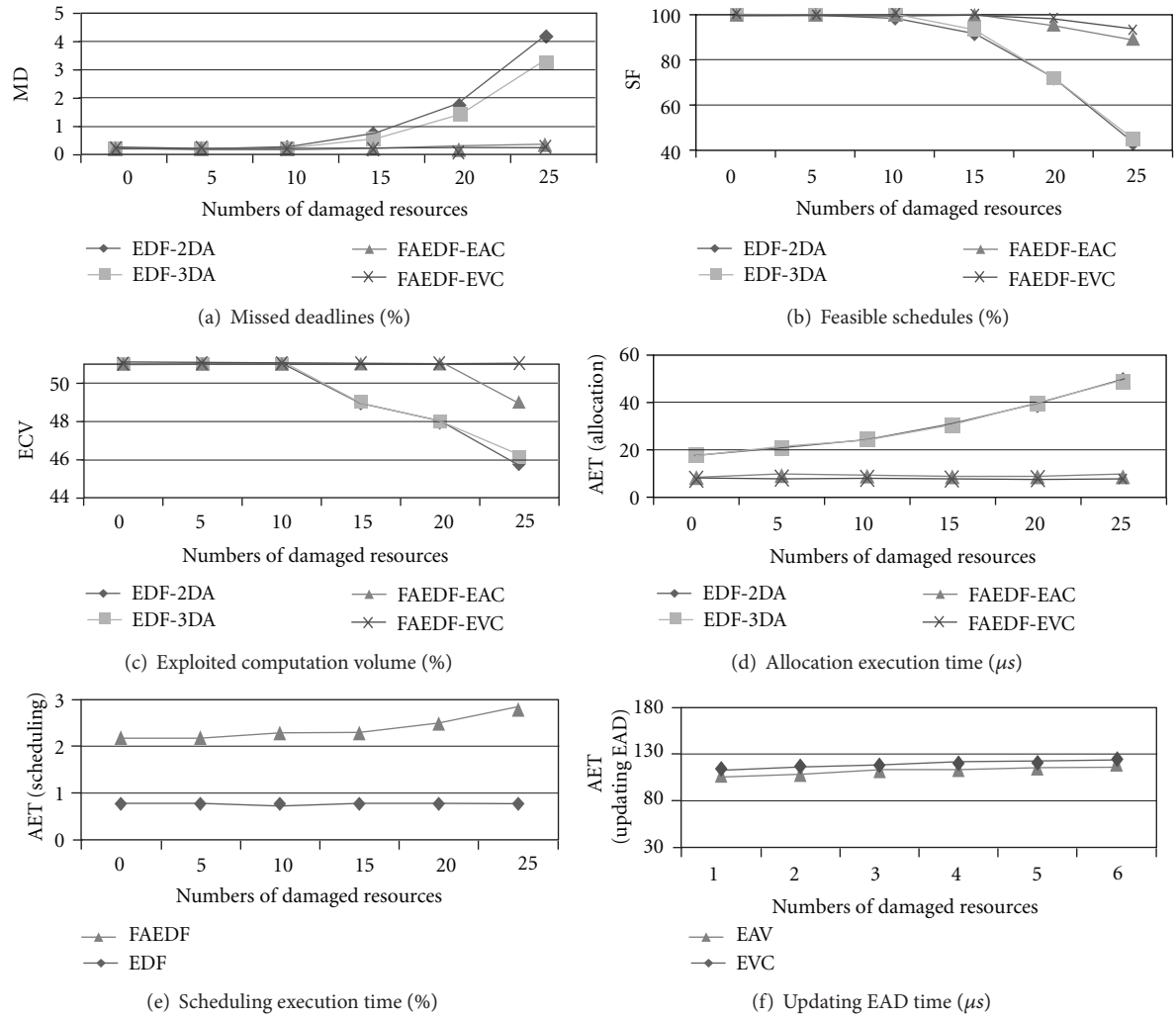



Figure 15: Medium utilization of the FPGA and moderate real-time requirements: $U_{\text{ICAP}} = 0.75$ and $U_{\text{COMP}} = 0.75$.

regard to 2DA/3DA as the FPGA gets more damage for most of the range of simulated faults. Finally, note that the performance of ECV and EAC is similar, with the former producing slightly better results.

Unlike MD, SF, and EVC, the execution time of the allocation algorithms, AET, shows a nearly linear increase with the number of simulated faults when using EAC/EVC heuristics. Indeed, the time needed to update the EAD in the worst situation is measured around 160 microseconds. Again, although the amount of time needed to make the scheduling decisions is always greater when using FAEDF than when using EDF, it is admissible (i.e., less than 3 microseconds). On the other hand, the time needed to allocate the tasks when using 2DA/3DA heuristics increases exponentially with the number of damaged resources in the chip, reaching up to 250 microseconds when the FPGA is highly utilized and significantly damaged (see Figure 17). Notably, this is even longer than the time needed to update the EAD in that situation. Hence, it cannot be claimed the online allocation capability for the 2DA/3DA heuristics when dealing with partially damaged FPGAs. Under the same conditions, note that the time needed to make the allocation decisions when using EAC/EVC heuristics is only 24 microseconds, which is an admissible overhead to target online task allocation. This important improvement is the result of the capability to discard unfeasible to allocate tasks early by both the scheduler (based on the MER size) and by the allocator (based on the `column_MER`s in the EAD). Finally, as expected, the execution time of ECV is slightly longer than that of EAC.

*7.4. Snake.* Figure 18 shows the results obtained when simulating the *Snake* approach on the realistic RC scenario described in Section 7.1. The results are normalised to the highest value, and in all of the cases FAEDF scheduling algorithm was used. We note that the fact of simulating most of the RC issues results in lower performance when using FAEDF-EAC and FAEDF-EVC than shown in previous simulations. For instance, the ECV was significantly smaller as a significant part of the sandbox could not be used to allocate HBC tasks due to fact that BRAMs are located in one edge. The sandbox is thus mainly used to allocate LBC tasks,

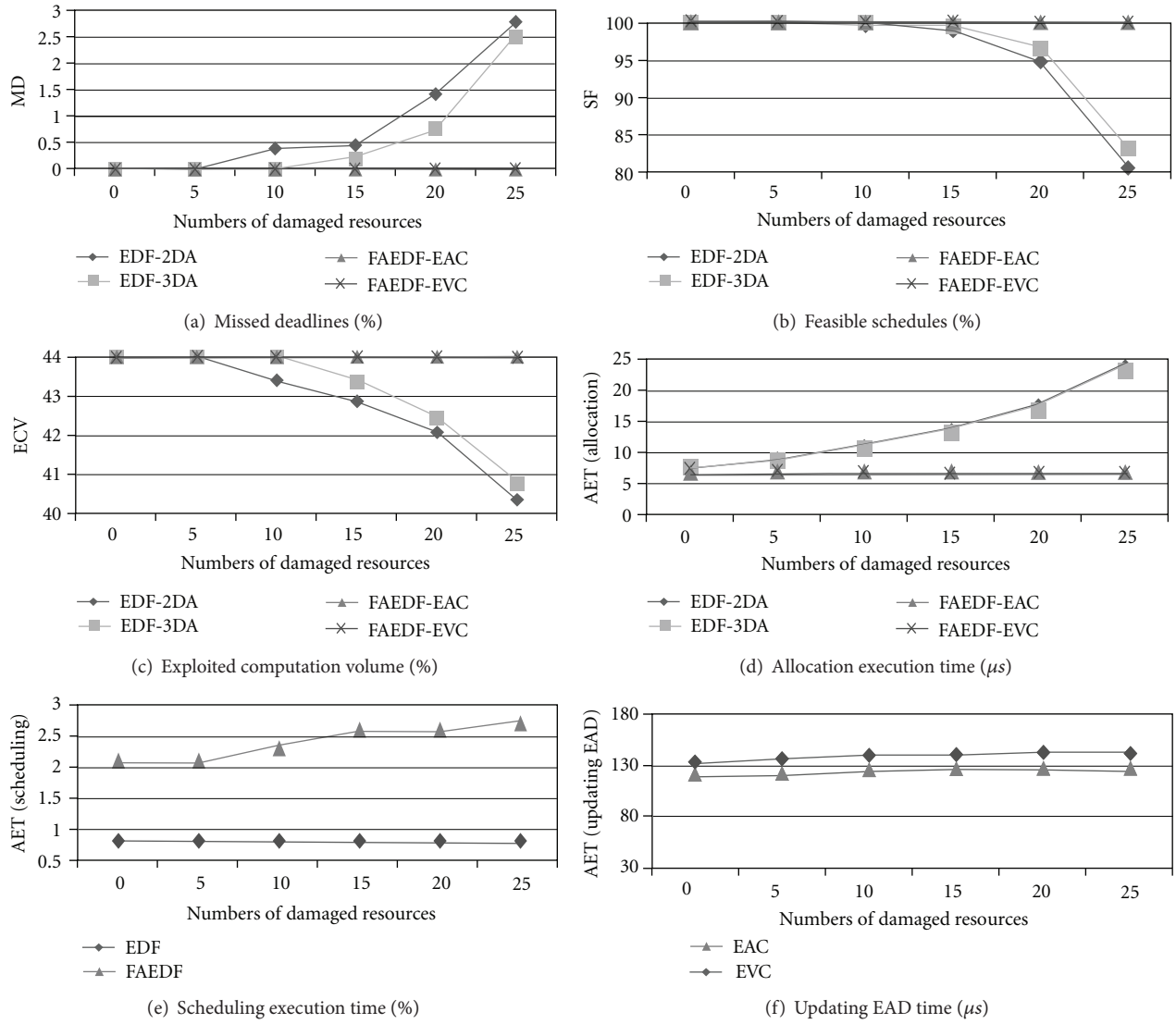



Figure 16: Low utilization of the FPGA and tight real-time requirements: $U_{\text{ICAP}} = 0.9$ and $U_{\text{COMP}} = 0.5$.

which represent a small fraction of the total amount of tasks generated.

Based on the obtained results, we conclude that *Snake* improves the performance shown when exclusively using EAC/EVC heuristics. This is reasonable as it is, indeed, especially conceived to extend these heuristics to deal with the simulated RC particularities and issues in this experiment (e.g., intertask dependencies and communications).

An important aspect to note is that the average time spent when making the allocation decisions in *Snake* is considerably reduced, as there is no need to evaluate all of the feasible allocations on the FPGA. When the tasks are reused no allocation, decisions must be made, and when the partial results are reused, only one allocation must be evaluated (for each version of the task). Moreover, using DRTs involves evaluating only a few more allocations, namely, those where DRTs are able to move input data from the data producer's ODB. In our simulations a maximum of 26 target allocations are considered when using DRTs: up to 4 clock regions above and below the ODB where the data is held, in the same BRAM column as well as in the neighbor right and left columns.

## 8. Implementation Details

A proof-of-concept R3TOS implementation has been developed on a Xilinx XC4VLX160 FPGA. As shown in Figure 19, the system comprises three main components: (1) a scheduler, (2) an allocator, and (3) a configuration manager to translate the high-level operations dictated by the scheduler and allocator into reconfiguration commands for the FPGA.

Each component is separately implemented to enable parallelism in the execution of the R3TOS processes. The parallel cooperation of simple components does not only result in low runtime overhead but also result in acceptable area overhead; that is, the main core of all R3TOS components is a tiny Xilinx PicoBlaze, which requires only 96 FPGA

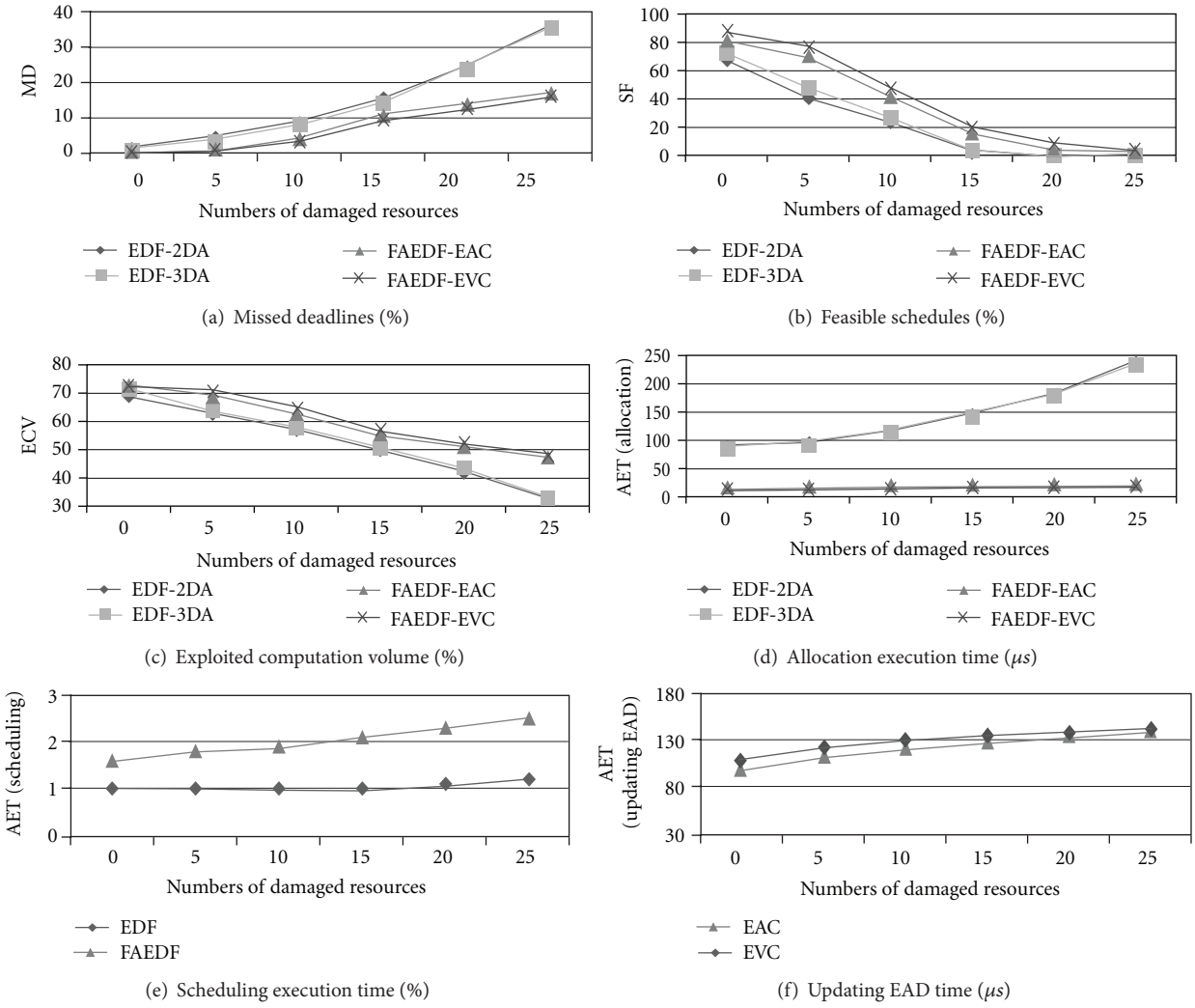



Figure 17: High utilization of the FPGA and loose real-time requirements: $U_{\text{ICAP}} = 0.5$ and $U_{\text{COMP}} = 0.9$.

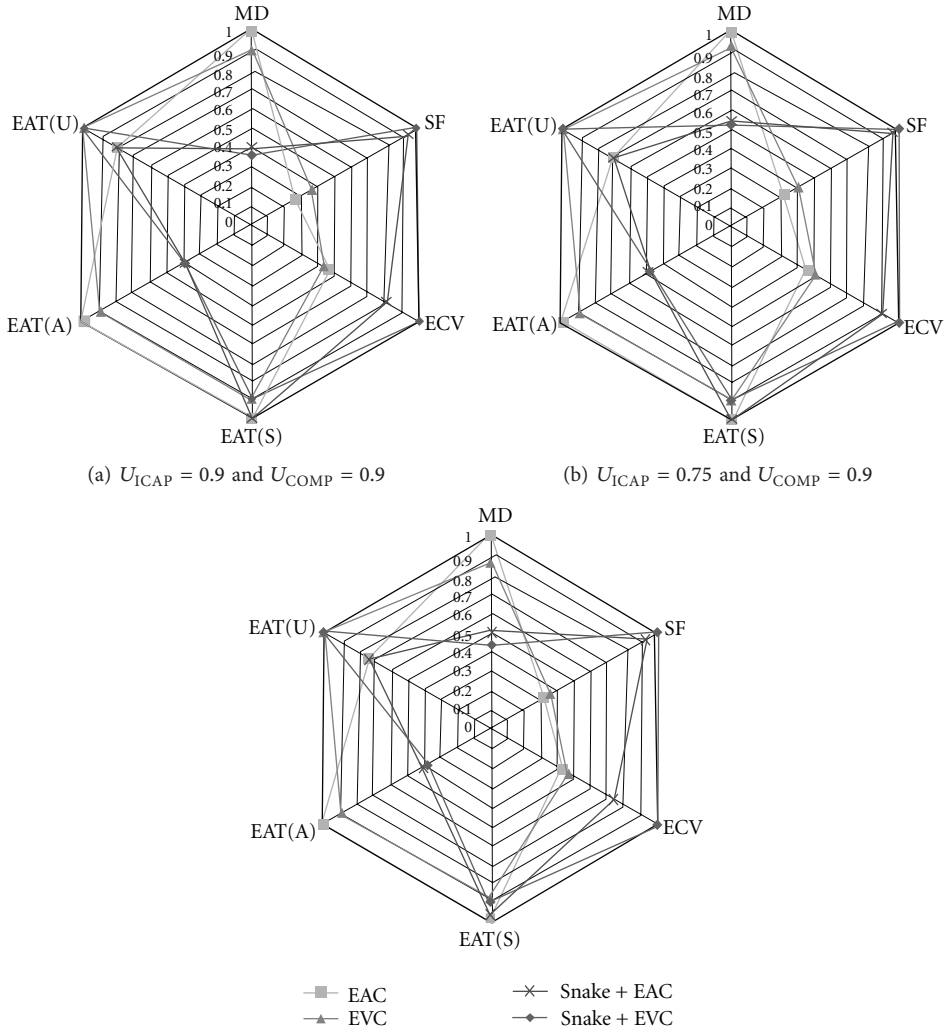
slices. Note that this architecture promotes upgradability, for example, the allocator and scheduler can be updated to run more efficient algorithms which might be designed in the future without having to modify the rest of components, and scalability, for example, multiple instances of the allocator can be used to speed up the allocation process in very large FPGAs. The cooperation among the R3TOS components is mastered by the scheduler, with the allocator and the configuration manager acting as slaves.

The internal architecture of the R3TOS components is structured around the PicoBlaze core. The PicoBlaze executes an optimized assembly program which is based on interruptions to reduce the response time, relying on an interrupt controller to handle the interruptions. Furthermore, each PicoBlaze uses a dedicated data BRAM to store the information associated with the corresponding R3TOS process(es) it executes. Hence, the scheduler manages the task queues in the task BRAM, the allocator keeps track of the available resources on an FPGA BRAM (state BRAM), where all of the area matrices presented in Section 6 are held in separate segments, and the configuration manager executes predefined sequences of configuration commands from a bitstream BRAM.

The configuration manager interacts with the configuration-related built-in logic included in the FPGA. Notably, it is equipped with specific hardware to drive the ICAP at the highest allowed clock frequency, achieving up to 390 MB/s reconfiguration throughput.

Specific for the scheduler is a timer to generate the kernel ticks $t_{\text{KT}}$. Additionally, the kernel timer supervises the correct functioning of the scheduler; that is, it acts as watchdog timer. The scheduler's PicoBlaze must generate at least one `alive` pulse within a maximum number of kernel ticks. Indeed, it is crucial for the reliability of the system to monitor the state of the scheduler as it is the master.

Specific for the allocator are three coprocessors, which are explained in the subsequent sections: (1) an Architecture Checker (AC) to speed up the search of feasible allocations where the FPGA layout is compatible with the internal architecture of the tasks, (2) an Empty Area Descriptor




(a) $U_{\text{ICAP}} = 0.9$ and $U_{\text{COMP}} = 0.9$

(b) $U_{\text{ICAP}} = 0.75$ and $U_{\text{COMP}} = 0.9$

(c) $U_{\text{ICAP}} = 0.9$ and $U_{\text{COMP}} = 0.75$

Figure 18: Performance on a realistic RC simulation scenario when using the *Snake* approach. EAT(U) refers to the EAD updating time, EAT(A) refers to the time needed for making the allocation decisions, and EAT(S) is the time needed for making the scheduling decisions.

Updater (EADU) to accelerate the intermediate computations required by the allocation algorithm, and (3) an Allocation Quality Evaluator (AQE) to accelerate the making of the allocation decisions. We resort to using the EAC heuristic as we believe it is simpler to implement than EVC and still produces good quality results.

*Empty Area Descriptor Updater (EADU)*. As previously introduced, the amount of time available to update the EAD without degrading the performance is limited by the duration of the set up phase of the tasks; that is, EAD is to be updated in parallel with task setting-up through the ICAP. In order to speed up the EAD updating, R3TOS includes a specific logic (EAD updater) that is coupled to the two ports of the FPGA State BRAM. This logic is very easy to control with only two interface signals: input `start` updating and output `end` updating. Both signals are driven by allocator's PicoBlaze, which also controls access to the FPGA State BRAM. Indeed, the latter BRAM is shared between the PicoBlaze, EADU, AQE, and configuration manager, which provides the list of detected damaged resources in the chip upon request by the allocator's Picoblaze. Prior to toggling the EADU, the `FPGA_state` is renewed by the allocator's PicoBlaze.

The EADU proceeds in four phases as shown in Figure 20. In the first phase, the `RAM` and `LAM` matrices are computed using the information included in the `FPGA_state` memory segment. These matrices are computed in parallel using the double port of the BRAM. As the only difference when computing the `RAM` and `LAM` matrices is that the `FPGA_state` is scanned in opposite directions, that is, right to left or left to right, the amount of time needed to compute both matrices is the same. Afterwards, the four area matrices `ULAM`, `DLAM`, `URAM`, and `DRAM` are computed in pairs using the two ports of the BRAM and the information included in the `RAM` and `LAM` memory segments. Namely, while the `ULAM` matrix is computed through the port A, the `URAM`

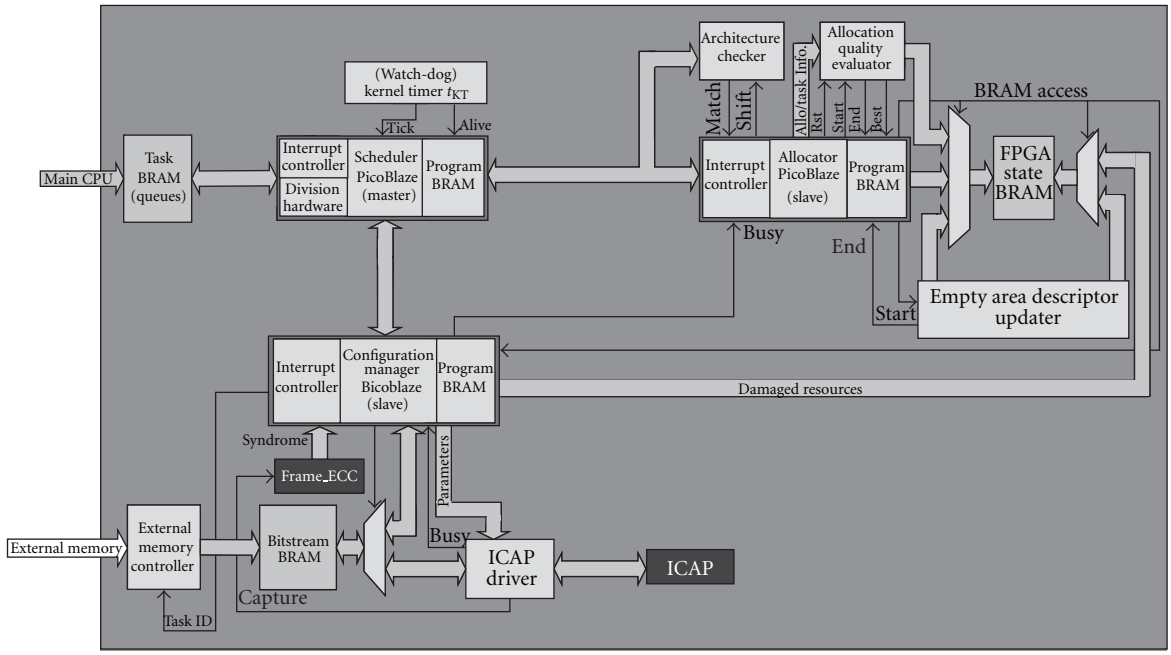



Figure 19: Overview of R3TOS implementation.

matrix is computed through the port B, and, then, DLAM and DRAM matrices are simultaneously computed through each BRAM port. Note that when computing the DRAM matrix, the column_MERs are also updated. In the fourth and last phase, the 2DAM matrix is updated using the area matrices computed during the previous three phases. Notably, this computation is speeded up by 2, as the values of two area matrices can be simultaneously accessed through the two ports of the FPGA State BRAM.

The EADU consumes 284 slices in the FPGA, and, in the situation described in Section 7, that is, $H_x = 15$ and $H_y = 12$, it allows for up to 34x speed improvement with regard to the solely PicoBlaze-based software implementation reported in [41].

*Architecture Checker (AC).* Figure 21 shows the structure of the AC, whose main objective is to rapidly check whether it is feasible to allocate a task on a given FPGA location in terms of types of resources, saving much computational cost to the allocator's PicoBlaze. The latter controls the AC block by means of two signals: `shift` and `match`. Note that this block is mainly used to find feasible allocations when systematically evaluating all of the candidate positions. On the other hand, when using *Snake*, the compatibility of FPGA's layout and task's internal architecture is checked by the PicoBlaze as the number of allocations to evaluate is reduced.

The central part of the AC module is a shift register (`ad`) of depth equal to $H_x$, that is, amount of columns in the FPGA. Another register (`AD`) is used to store the architecture descriptor of the FPGA device itself. Each cell in these registers accounts thus for a resource column, being the type of resource coded using 2 bits: "00" for CLBs, "01" for DSP48s, "10" for BRAMs, and "11" for other resources (e.g., PowerPC and IOBs). The architecture descriptor of the task to check $\theta_i$ is loaded in the `ad` register and shifted cell by cell to cover all of the possible allocations for it, that is, until the task descriptor reaches the deepest $H_x$ cell in the shift register. To check whether the type of resources required by the task matches with the FPGA resources actually available in each position, both `ad` and `AD` cells are XORed, where "0" means that the type of all of the resources is the same. The cells are individually enabled to take part in the XOR operation in order to exclude the resources which are not actually used by the task in each checked position. To do this a shift register (`EN`) with the same depth of `ad` and `AD` is used. This register is initially loaded with all zeros, except for the cells occupied by the task descriptor, which are loaded with "1"; that is, $\texttt{EN}(i) =$ "1" for all $\leq h_{x,i}$, and $\texttt{EN}(i) =$ "0" for all $i > h_{x,i}$. The sequence of "1"s is shifted in the `EN` register when the task descriptor is shifted in the `ad` register to reflect which columns are to be used by the task in the checked position at any time. Therefore, the PicoBlaze is only responsible for controlling the shift in the registers and checking the feasibility of the placement. The latter is given by the `match` signal, where `match` = "1" means that allocation is feasible.

The AC requires about half of the slices required by a PicoBlaze, namely, 46 slices. The upper bound speed improvement brought about by this module when checking the allocatability of a task $\theta_i$ compared to a solely PicoBlaze-based software implementation can be roughly estimated to be around $2 \cdot h_{x,i}x$. Indeed, without using the AC, 6 PicoBlaze instructions are needed to check the resource compatibility in each FPGA column: 2 instructions for accessing the FPGA descriptor, another 2 instructions for accessing the task descriptor, 1 instruction to compare both values, and another instruction to update the next column to be checked. On the

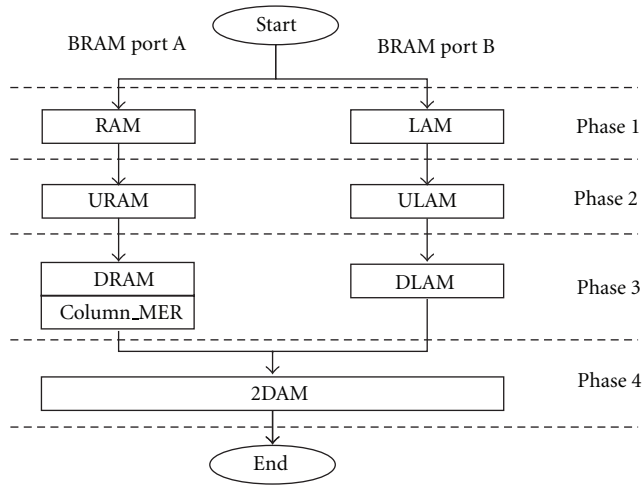



Figure 20: EAD updating process.

other hand, when using the AC, only 3 PicoBlaze instructions are required to perform this check: 2 instructions to enable and disable the `shift` signal and another instruction to input the `match` value. Assuming the worst case where all resource columns are compatible, the achieved 2x speed-up factor brought about by the AC when checking the resource compatibility of one column is extended to the $h_{x,i}$ columns the task spans.

*Allocation Quality Evaluator (AQE).* When toggled by the allocator's PicoBlaze, the AQE takes over access to the FPGA State BRAM to quickly compute the quality of candidate allocations. This is done by adding the EAC scores stored in the `2DAM` segment that correspond to the FPGA positions to be assigned to the task in each evaluated allocation. The latter computation is automatically performed by the AQE based on the allocation and task information passed by the PicoBlaze, that is, $h_{x,i}$, $h_{y,i}$, $X_{\text{allo}}$, and $Y_{\text{allo}}$. Besides, the AQE keeps track of the quality of the checked allocations, indicating to the PicoBlaze which is the one that produces the least fragmentation on the FPGA by means of the `best` signal (i.e., the lowest sum of EAC scores). It is important to note that the AQE does not check the feasibility of the allocations. This is done by the allocator's PicoBlaze by consulting the EAD and using the AC.

The interface of the AQE consists of 4 control signals (`rst`, `start`, `end`, and `best`) as well as an 8-bit input to receive the allocation and task parameters.

The AQE consumes 33 slices in the FPGA and allows for a significant speed improvement when making the allocation decisions with regard to the solely PicoBlaze-based software implementation reported in [41]. The achievable acceleration increases with the size of the task to allocate until it reaches a high bound of about 9x. This behaviour is due to two reasons. First, when dealing with big tasks, the communication overhead between PicoBlaze and AQE is smaller because there are fewer candidate allocations to check. Second, evaluating the allocation quality of a big task requires more computations (i.e., additions) to be done, which are indeed accelerated by the AQE.

Table 2: Measured performance figures.

| | Min. | Max. |
|---|---|---|
| Scheduling | | |
| Scheduling algorithm execution | <1 $\mu$s | 100 $\mu$s |
| Queues and task state updating | <1 $\mu$s | 300 $\mu$s |
| Allocation | | |
| Allocation algorithm execution | <1 $\mu$s | 100 $\mu$s |
| Empty Area Descriptor updating | 10 $\mu$s | 200 $\mu$s |
| Inter-task communications | | |
| Transfer LUT data buffer (ICAP) | 3.7 $\mu$s | 3.7 $\mu$s |
| Transfer BRAM data buffer (ICAP) | 60.18 $\mu$s | 60.18 $\mu$s |
| Transfer BRAM data buffer (DRTS): configuration layer | 36.18 $\mu$s | 39.9 $\mu$s |
| Transfer BRAM data buffer (DRTS): functional layer | 81.92 $\mu$s | 81.92 $\mu$s |
| Switching brams between neighbor tasks | 10.03 $\mu$s | 10.03 $\mu$s |
| Inter-task synchronization | | |
| Polling of a HWS | 1.6 $\mu$s | 1.6 $\mu$s |
| Activation of a HWS | 3.7 $\mu$s | 3.7 $\mu$s |

*8.1. Performance Evaluation.* Table 2 shows the most significant performance figures measured in the developed proof-of-concept R3TOS implementation when it runs with a 100 MHz clock. Notably, the amount of time needed by the scheduler and allocator to update the task queues and EAD is slightly shorter (i.e. hundreds of microseconds) of that needed to set-up a typical hardware task using the ICAP (usually several hundreds of microseconds or few milliseconds [42]), making it possible to reduce the time overheads introduced by R3TOS as these three processes can be concurrently carried out in most of the cases. The obtained results are promising in light of enabling the use of our solution with newer reconfigurable technology with presumably faster reconfiguration capabilities and larger sizes.

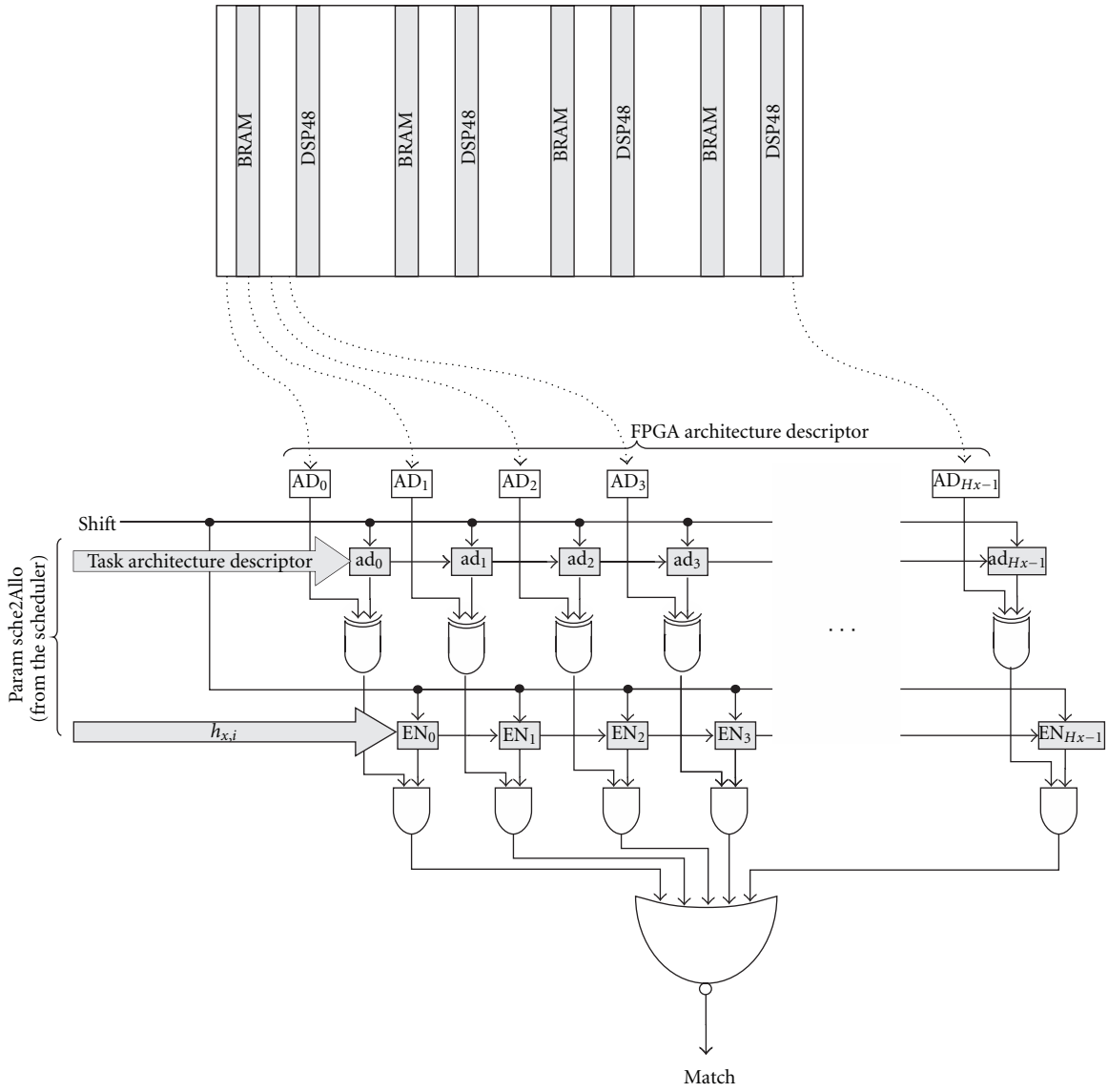



Figure 21: Simplified structure of the Architecture Checker (AC).

We acknowledge the existence of a time overhead which is introduced by R3TOS when making the scheduling and allocation decisions. Since these decisions are mostly made by software routines in the allocator and scheduler PicoBlazes, the overhead can reach up to tens of microseconds per each task allocation attempt. Note that although the allocator PicoBlaze relies on the AQE to accelerate the allocation process, it is still responsible for exploring the EAD to find feasible allocation candidates to be evaluated, thus limiting the achievable acceleration. As for the allocator, it would be convenient to use a hardware accelerator in the scheduler when dealing with a large number of tasks in order to keep the task management overheads within reasonable bounds.

Table 2 also shows the achievable acceleration when exchanging data among tasks using DRTs or directly accessing the data the data in producer task's ODB. While the time needed to transfer the content of a BRAM-based data buffer using the ICAP is about 60 microseconds, the access to the BRAM can be switched from the producer to the consumer task in only 10.03 microseconds (around 6x speed-up). In addition, 36.18 microseconds are needed to configure a DRT (around 1.6x speed-up), which then requires 81.92 microseconds to complete the data transfer through the functional layer. Note that the latter time does not constrain the performance as it can be parallelized with the task set-up phase.

## 9. Conclusions

In this article a novel scheduling algorithm and two novel allocation heuristics have been presented in the scope of our R3TOS project. First, the Finishing Aware EDF (FAEDF) scheduling algorithm improves nonpreemptive EDF by delaying the execution of tasks which cannot be allocated in

the first instance until enough adjacent free area is released on the FPGA. Second, the Empty Area/Volume Compaction (EAC/EVC) heuristics outperform related work in the field, especially when the FPGA is partially damaged. Finally, the *Snake* task allocation strategy has been introduced. This novel approach promotes the concatenation of tasks, as the input edge of one task can be placed next to the output edge of the previously executed task in the pipeline in such a way that memory elements where data to be exchanged is kept are switched between both tasks. The proposed algorithms and strategies are proven to be efficacious by means of synthetic simulations, and their runtime execution overhead measured in a real hardware implementation is proven to be admissible. The latter hardware implementation, which also includes specific circuitry to drive the ICAP at its highest rate, has a relatively small footprint: 2,003 slices and 6 BRAMs (around 5% of the logic resources and 2% of the storage resources of an XC4VLX160 FPGA). Future work targets the evaluation of these approaches in a real-world application.

## References


[1] G. J. Brebner, "A virtual hardware operating system for the Xilinx XC6200," in *Proceedings of the International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 327–336, 1996.

[2] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[3] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann, San Francisco, Calif, USA, 1st edition, 2007.

[4] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[5] D. P. Montminy, R. O. Baldwin, P. D. Williams, and B. E. Mullins, "Using relocatable bitstreams for fault tolerance," in *Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS '07)*, pp. 701–708, August 2007.

[6] X. Iturbe, K. Benkrid, T. Arslan, I. Martinez, M. Azkarate, and A. Morales-Reyes, "Evolutionary dynamic allocation of relocatable modules onto partially damaged Xilinx FPGAs," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pp. 211–217, 2010.

[7] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, and I. Martinez, "Empty resource compaction algorithms for real-time hardware tasks placement on partially reconfigurable FPGAs subject to fault occurrence," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs*, 2011.

[8] X. Iturbe, K. Benkrid, A. Ebrahim, C. Hong, T. Arslan, and I. Martinez, "Snake: an ecient strategy for the reuse of circuitry and partial computation results in high-performance recongurable computing," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '11)*, 2011.

[9] K. Danne and M. Platzner, "Periodic real-time scheduling for FPGA computers," in *3rd International Workshop on Intelligent Solutions in Embedded Systems (WISES '05)*, pp. 117–127, May 2005.

[10] . Steiger, H. Walder, and M. Platzner, "Heuristics for online scheduling real-time tasks to partially reconfigurable devices," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 575–584, 2003.

[11] Y.-H. Chen and P.-A. Hsiung, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC," in *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pp. 489–498, 2005.

[12] X. G. Zhou, Y. Wang, X. Z. Huang, and C. L. Peng, "On-line scheduling of real-time tasks for reconfigurable computing system," in *Proceedings of the IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 57–64, December 2006.

[13] X. Zhou, Y. Wang, X. Huang, and C. Peng, "Fast on-line task placement and scheduling on reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 132–138, August 2007.

[14] J. Cui, Z. Gu, W. Liu, and Q. Deng, "An efficient algorithm for online soft real-time task placement on reconfigurable hardware devices," in *Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07)*, pp. 321–328, May 2007.

[15] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 123–128, April 2007.

[16] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "Online task scheduling for the FPGA-based partially reconfigurable systems," in *Proceedings of the International Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, pp. 216–230, 2009.

[17] T. Marconi, Y. Lu, K. Bertels, and G. Gaydadjiev, "3D compaction: a novel blocking-aware algorithm for online hardware task scheduling and placement on 2D partially reconfigurable devices," in *Proceedings of the International Symposium on Appied Reconfigurable Computing*, pp. 194–206, 2010.

[18] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "A communication aware online task scheduling algorithm for FPGA-based partially reconfigurable systems," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 65–68, May 2010.

[19] D. Ghringer, M. Hübner, E. Nguepi Zeutebouo, and J. Becker, "Operating system for runtime reconfigurable multiprocessor systems," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 121353, 16 pages, 2011.

[20] F. Redaelli, M. D. Santambrogio, and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 97–102, mex, December 2008.

[21] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[22] H. Walder, C. Steiger, and M. Platzner, "Fast online task placement on FPGAs: free space partitioning and 2D-hashing," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[23] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pp. 1825–1831, April 2004.

[24] M. Handa and R. Vemuri, "An efficient algorithm for finding empty space for online FPGA placement," in *Proceedings of the 41st Design Automation Conference*, pp. 960–965, June 2004.

[25] M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto, "Core allocation and relocation management for a self dynamically reconfigurable architecture," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI: Trends in VLSI Technology and Design (ISVLSI '08)*, pp. 286–291, April 2008.

[26] M. Tomono, M. Nakanishi, S. Yamashita, K. Nakajima, and K. Watanabe, "A new approach to online FPGA placement," in *Proceedings of the 40th Annual Conference on Information Sciences and Systems (CISS '06)*, pp. 145–150, March 2006.

[27] A. Ahmadinia, C. Bobda, S. P. Fekete, J. Teich, and J. C. van der Veen, "Optimal free-space management and routing-conscious dynamic placement for reconfigurable devices," *IEEE Transactions on Computers*, vol. 56, no. 5, pp. 673–680, 2007.

[28] J. Tabero, J. Septien, H. Mecha, and D. Mozos, "A low fragmentation heuristic for task placement in 2D RTR HW management," in *Proceedings of the International Conference on Field-Programmable Logic and Applications*, pp. 241–250, 2004.

[29] J. Tabero, J. Septién, H. Mecha, and D. Mozos, "Task placement heuristic based on 3D-adjacency and look-ahead in reconfigurable systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '06)*, pp. 396–401, January 2006.

[30] J. Tabero, J. Septién, H. Mecha, and D. Mozos, "Allocation heuristics and defragmentation measures for reconfigurable systems management," *Integration, the VLSI Journal*, vol. 41, no. 2, pp. 281–296, 2008.

[31] C.-H. Lu, H.-W. Liao, and P.-A. Hsiung, "Multi-objective placement of reconfigurable hardware tasks in real-time system," in *Proceedings of the International Conference on Computational Science and Engineering*, pp. 921–925, 2009.

[32] Z. Li and S. Hauck, "Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation," in *Proceedings of the 10th ACM International Symposium on Field-Programmable Gate Arrays (FPGA '02)*, pp. 187–195, February 2002.

[33] A. Ejnioui and R. F. DeMara, "Area reclamation strategies and metrics for SRAM-based reconfigurable devices," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05)*, pp. 196–202, June 2005.

[34] J. C. Van Der Veen, S. P. Fekete, M. Majer et al., "Defragmenting the module layout of a partially reconfigurable device," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 92–101, June 2005.

[35] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 223–228, August 2005.

[36] A. A. El Farag, H. M. El-Boghdadi, and S. I. Shaheen, "Improving utilization of reconfigurable resources using two-dimensional compaction," *Journal of Supercomputing*, vol. 42, no. 2, pp. 235–250, 2007.

[37] A. Ahmadinia, C. Bobda, M. Bednara, and J. Teich, "A new approach for on-line placement on reconfigurable devices," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, pp. 1825–1831, April 2004.

[38] A. Montone, F. Redaelli, M. D. Santambrogio, and S. O. Memik, "A reconfiguration-aware floorplacer for FPGAs," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '08)*, pp. 109–114, December 2008.

[39] S. Srinivasan, P. Mangalagiri, Y. Xie, N. Vijaykrishnan, and K. Sarpatwari, "FLAW: FPGA lifetime awareness," in *Proceedings of the Annual Design Automation Conference*, pp. 630–635, 2006.

[40] X. Iturbe, K. Benkrid, T. Arslan, C. Hong, and I. Martinez, "Enabling FPGAs for future deep space exploration missions: improving fault-tolerance and computation density with R3TOS," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, 2011.

[41] C. Hong, K. Benkrid, X. Iturbe, A. Ebrahim, and T. Arslan, "Efficient on-chip task scheduler and allocator for reconfigurable operating systems," *Embedded Systems Letters*, vol. 3, no. 3, pp. 85–88, 2011.

[42] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 498–502, September 2009.